一、论文概况

Abstract

MAMADROID: 通过构建行为模型的马尔可夫链检测Android恶意软件

摘要: Android平台的普及使得针对其的恶意软件威胁迅猛增长。由于Android恶意软件和操作系统本身不断演变，设计能够长期运行而无需修改或昂贵重新训练的强大恶意软件防护技术非常具有挑战性。在本文中，我们提出了MAMADROID，一种依赖于应用程序行为的Android恶意软件检测系统。MAMADROID从应用程序执行的抽象API调用的序列中构建行为模型，以此来提取特征并进行分类。通过将调用抽象为其包或家族，MAMADROID能够保持对API更改的适应性并保持特征集大小可管理。我们在一个包含8.5K个良性应用和35.5K个恶意应用的数据集上评估其准确性，结果显示它不仅能有效检测恶意软件（F-measure高达99%），而且系统构建的模型在长时间内保持其检测能力（平均分别为87%和73%的F-measure，即训练后一年和两年）。最后，我们与依赖应用程序API调用频率的先进系统DROIDAPIMINER进行比较，结果显示MAMADROID明显优于其。

二、论文翻译

I. INTRODUCTION Part-1

In the first quarter of 2016, 85% of smartphone sales were devices running Android [49]. Due to its popularity, cybercriminals have increasingly targeted this ecosystem [17], as malware running on mobile devices can be particularly lucrative -e.g., allowing attackers to defeat two factor authentication [51], [53] or trigger leakage of sensitive

I. INTRODUCTION

在2016年第一季度，85%的智能手机销售量是安卓设备[49]。由于其受欢迎程度，网络犯罪分子越来越多地针对这个生态系统进行攻击[17]，因为在移动设备上运行的恶意软件可能特别有利可图 - 例如，允许攻击者击败双因素身份验证[51]，[53]或触发敏感信息泄漏[27]。在移动设备上检测恶意软件比在台式机/笔记本电脑上要面临额外的挑战：智能手机的电池寿命有限，使用需要持续扫描和复杂计算的传统方法是不可行的[43]。因此，安卓恶意软件的检测通常由谷歌在集中式的方式下

information [27]. Detecting malware on mobile devices presents additional challenges compared to desktop/laptop computers: smartphones have limited battery life, making it infeasible to use traditional approaches requiring constant scanning and complex computation [43]. Therefore, Android malware detection is typically performed by Google in a centralized fashion, i.e., by analyzing apps submitted to the Play Store using a tool called Bouncer [40]. However, many malicious apps manage to avoid detection [1], and anyway Android's openness enables manufacturers and users to install apps that come from third-party market places, which might not perform any malware checks at all, or anyway not as accurately [67]. As a result, the research community has devoted significant attention to malware detection on Android. Previous work has often relied on the permissions requested by apps [20], [46], using models built from malware samples. This strategy, however, is prone to false positives, since there are often legitimate reasons for benign apps to request permissions classified as dangerous [20]. Another approach, used by DROIDAPIMINER [2], is to perform classification based on API calls frequently used by malware. However, relying on the most common calls observed during training prompts the need for constant retraining, due to the evolution of malware and the Android API alike. For instance, "old" calls are often deprecated with new API releases, so malware developers may switch to different calls to perform similar actions, which affects DROIDAPIMINER's effectiveness due to its use of specific calls. In this paper, we present a novel malware detection system for Android that instead relies on the sequence of abstracted API calls performed by an app rather than their use or frequency, aiming

进行，即通过使用名为Bouncer的工具分析提交到Play商店的应用程序[40]。然而，许多恶意应用程序设法避开了检测[1]，而且安卓的开放性使制造商和用户能够安装来自第三方市场的应用程序，这些应用程序可能根本不执行任何恶意软件检查，或者不如谷歌的准确[67]。

因此，研究界对安卓恶意软件检测倾注了大量注意力。以前的工作通常依赖于应用程序请求的权限[20]，[46]，使用从恶意软件样本构建的模型。然而，这种策略容易产生误报，因为对于被分类为危险的权限，通常存在合理的原因使得良性应用请求这些权限[20]。另一种方法是由DROIDAPIMINER [2]使用的方法，即根据恶意软件频繁使用的API调用进行分类。然而，依赖于训练中观察到的最常用的调用会导致需要不断重新训练，因为恶意软件和安卓API都在不断演化。例如，“旧的”调用在新的API版本中通常被废弃，因此恶意软件开发者可能会转而使用不同的调用来执行类似的操作，这会影响DROIDAPIMINER的有效性，因为它使用了特定的调用。

在本文中，我们提出了一种新颖的安卓恶意软件检测系统，该系统不依赖于应用程序使用或频率，而是以应用程序执行的抽象API调用序列为基础，旨在捕捉应用程序的行为模型。我们称之为MAMADROID的系统，将API调用抽象为调用的包名（例如java.lang）或其来源（例如java，android，google），我们称之为家族。抽象可以使系统对安卓框架中的API更改具有弹性，因为家族和包名的添加和删除频率低于单个API调用。同时，这种抽象并不模糊应用程序的行为：例如，包名包括用于对相似对象执行相似操作的类和接口，因此我们可以从包名中对操作类型进行建模，而与底层的类和接口无关。例如，我们知道java.io包用于系统I/O和访问文件系统，尽管包中提供了用于此类操作的不同类和接口。

to capture the behavioral model of the app. Our system, which we call MAMADROID, abstracts API calls to either the package name of the call (e.g., java.lang) or its source (e.g., java, android, google), which we refer to as family. Abstraction provides resilience to API changes in the Android framework as families and packages are added and removed less frequently than single API calls. At the same time, this does not abstract away the behavior of an app: for instance, packages include classes and interfaces used to perform similar operations on similar objects, so we can model the types of operations from the package name, independently of the underlying classes and interfaces. For example, we know that the java.io package is used for system I/O and access to the file system, even though there are different classes and interfaces provided by the package for such operations.

I. INTRODUCTION Part-2

After abstracting the calls, MAMADROID analyzes the sequence of API calls performed by an app, aiming to model the app's behavior. Our intuition is that malware may use calls for different operations, and in a different order, than benign apps. For example, android.media.MediaRecorder can be used by any app that has permission to record audio, but the call sequence may reveal that malware only uses calls from this class after calls to getRunningTasks(), which allows recording conversations [65], as opposed to benign apps where calls from the class may appear in any order. Relying on the sequence of

I. INTRODUCTION

在调用的基础上进行抽象后，MAMADROID分析应用程序执行的API调用序列，旨在建模应用程序的行为。我们的直觉是，与良性应用程序相比，恶意软件可能以不同的顺序和方式使用调用进行不同的操作。例如，android.media.MediaRecorder可以被任何具有录音权限的应用程序使用，但调用序列可能显示，恶意软件仅在调用getRunningTasks（）之后才使用来进行录音对话 [65]，而良性应用程序中该类调用可按任何顺序出现。通过依赖抽象调用的序列，我们可以以比之前的工作更复杂的方式建模行为，而之前的工作只关注特定API调用或权限的存在与否[2]，[5]，同时使问题可控[33]。MAMADROID 构建了一个统计模型来表示应用程序执行的API调用之间的转换，具体来说，我们将这些转换模型化为马尔可夫链，并使用它们提取特征并进行分类，即将应用程序

abstracted calls allows us to model behavior in a more complex way than previous work, which only looked at the presence or absence of certain API calls or permissions [2], [5], while still keeping the problem tractable [33]. MAMADROID builds a statistical model to represent the transitions between the API calls performed by an app, specifically, we model these transitions as Markov chains, and use them to extract features and perform classification (i.e., labeling apps as benign or malicious). Calls are abstracted to either their package or their family, i.e., MAMADROID operates in one of two modes, depending on the abstraction granularity. We present a detailed evaluation of both classification accuracy (using F-measure, precision, and recall) and runtime performance of MAMADROID, using a dataset of almost 44K apps (8.5K benign and 35.5K malware samples). We include a mix of older and newer apps, from October 2010 to May 2016, verifying that our model is robust to changes in Android malware samples and APIs. To the best of our knowledge, this is the largest malware dataset used to evaluate an Android malware detection system in a research paper. Our experimental analysis shows that MAMADROID can effectively model both benign and malicious Android apps, and perform an efficient classification on them. Compared to other systems such as DROIDAPIMINER [2], our approach allows us to account for changes in the Android API, without the need to frequently retrain the classifier. We show that MAMADROID is able to effectively detect unknown malware samples not only in the "present," (with F-measure up to 99%) but also consistently over the years (i.e., when the system is trained on older samples and classification performed over newer ones), as it keeps an average detection

标记为良性或恶意。调用被抽象为其包或其系列，即，MAMADROID根据抽象的细粒度进行操作的方式可以是两种之一。 我们使用一个包含近44K个应用程序（8.5K个良性和35.5K个恶意样本）的数据集，对MAMADROID的分类准确性（使用F-measure、精确度和召回率）以及运行时性能进行了详细评估。我们混合了旧的和新的应用程序，从2010年10月到2016年5月，验证了我们的模型对Android恶意软件样本和API的变化具有鲁棒性。据我们所知，这是在研究论文中评估Android恶意软件检测系统时使用的最大恶意软件数据集。我们的实验分析表明，MAMADROID能够有效地建模良性和恶意的Android应用程序，并对它们进行高效的分类。与其他系统（如DROIDAPIMINER [2]）相比，我们的方法使我们能够处理Android API的变化，而无需频繁重新训练分类器。 我们展示了MAMADROID能够有效地检测未知的恶意软件样本，不仅在“当前”时期（F-measure高达99%），而且在多年期间保持一致（即在系统对旧样本进行训练并对新样本进行分类时），其平均检测准确率（以F-measure衡量）为一年后的87%和两年后的73%（而DROIDAPIMINER [2]仅达到46%和42%）。我们还强调，当系统不再有效（测试集比训练集新超过两年），这是由于MAMADROID具有较低的召回率，但保持较高的精确度。我们还进行相反的实验，即使用新样本进行训练，并验证系统仍然能够检测旧的恶意软件。这尤其重要，因为它显示了MAMADROID可以检测到新的威胁，同时仍然能够识别已经在野外存在一段时间的恶意软件样本。

accuracy, evaluated in terms of F-measure, of 87% after one year and 73% after two years (as opposed to 46% and 42% achieved by DROIDAPIMINER [2]). We also highlight that when the system is not efficient anymore (when the test set is newer than the training set by more than two years), it is as a result of MAMADROID having low recall, but maintaining high precision. We also do the opposite, i.e., training on newer samples and verifying that the system can still detect old malware. This is particularly important as it shows that MAMADROID can detect newer threats, while still identifying malware samples that have been in the wild for some time.

I. INTRODUCTION Part-3

Summary of Contributions. First, we introduce a novel approach, implemented in a tool called MAMADROID, to detect Android malware by abstracting API calls to their package and family, and using Markov chains to model the behavior of the apps through the sequences of API calls. Second, we can detect unknown samples on the same year of training with an F-measure of 99%, but also years after training the system, meaning that MAMADROID does not need continuous re-training. Our system is scalable as we model every single app independently from the others and can easily append app features in a new training set. Finally, compared to previous work [2], MAMADROID achieves significantly higher accuracy with reasonably fast running times, while also being more robust to evolution in malware development and changes in the Android API. Paper Organization. The rest of the

I. INTRODUCTION

I. 引言 第3部分：贡献摘要。

首先，我们介绍了一种新颖的方法，通过将API调用抽象为它们的包和家族，并使用马尔可夫链来模拟应用程序通过API调用序列的行为，实现了一个名为MAMADROID的工具，用于检测Android恶意软件。

其次，我们可以在训练的同一年中检测到未知样本，其F-measure达到99%，而且在训练系统后的多年内也可以进行检测，这意味着MAMADROID不需要进行持续的重新训练。

我们的系统可扩展，因为我们独立地对每个应用进行建模，与其他应用无关，并且可以在新的训练集中轻松添加应用特征。

最后，与之前的工作[2]相比，MAMADROID在合理的运行时间内实现了显著更高的准确性，同时对恶意软件开发和Android API的演变更具稳健性。

论文组织。本文的其余部分组织如下。下一节介绍了MAMADROID系统，然后第三节介绍了我们评估使用的数据集（第四节），而第五节进一步讨论了我们的结果以及其局限性。在第六节回顾相关工作后，本文在第七节总结。

paper is organized as follows. The next section presents the MAMADROID system, then, Section III introduces the datasets used in our evaluation (Section IV), while Section V further discusses our results as well as its limitations. After reviewing related work in Section VI, the paper concludes in Section VII.

?

Fig. 1: Overview of MAMADROID operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious.

?

图 1：MAMADROID操作概览。在（1）中，它从Android应用程序中提取调用图，接下来，它从调用图中构建（抽象化的）API调用序列（2）。在（3）中，调用序列被用来构建马尔可夫链和该应用程序的特征向量。最后，在（4）中进行分类，将该应用程序标记为良性或恶意。

A. Overview

We now introduce MAMADROID, a novel system for Android malware detection. MAMADROID characterizes the transitions between different API calls performed by Android apps -i.e., the sequence of API calls. It then models these transitions as Markov chains, which are in turn used to extract features for machine learning algorithms to classify apps as benign or malicious. MAMADROID does not actually use the sequence of raw API calls, but abstracts each call to either its package or its family. For instance, the API call getMessage() is parsed as: package java family .lang.Throwable: String getMessage()

A. Overview

我们现在介绍一种名为MAMADROID的新型Android恶意软件检测系统。MAMADROID通过对Android应用程序执行的不同API调用之间的转换进行特征化，即API调用序列。然后，它将这些转换建模为马尔可夫链，并将其用于提取特征，以供机器学习算法对应用程序进行分类，判断其是否良性或恶意。MAMADROID实际上并不使用原始API调用序列，而是将每个调用抽象为其包或其类别。例如，API调用getMessage()被解析为：包java类别.lang.Throwable: String getMessage()。

API call

Given these two different types of abstractions, we have two modes of operation for MAMADROID, each using one of the types of abstraction. We test both, highlighting their advantages and disadvantages -in a nutshell, the abstraction to family is more lightweight, while that to package is more fine-grained. MAMADROID's operation goes through four phases, as depicted in Fig. 1. First, we extract the call graph from each app by using static analysis (1), next we obtain the sequences of API calls for the app using all unique nodes in the call graph and associating, to each node, all its child nodes (2). As mentioned, we abstract a call to either its package or family. Finally, by building on the sequences, MAMADROID constructs a Markov chain model (3), with the transition probabilities used as the feature vector to classify the app as either benign or malware using a machine learning classifier (4). In the rest of this section, we discuss each of these steps in detail.

API call

根据这两种不同的抽象类型，MAMADROID共有两种操作模式，每种模式使用一种抽象类型。我们测试了这两种模式，并突出它们的优缺点 - 简言之，对于家族的抽象更轻量级，而对于包的抽象更细粒度。MAMADROID的操作经过四个阶段，如图1所示。首先，我们使用静态分析从每个应用程序中提取调用图（1），然后使用调用图中的所有唯一节点获取应用程序的API调用序列，并将每个节点的所有子节点关联起来（2）。正如前面提到的，我们将调用抽象为包或家族。最后，通过基于这些序列构建Markov链模型（3），MAMADROID使用转移概率作为特征向量，通过机器学习分类器将应用程序分类为良性或恶意（4）。在本节的其余部分，我们详细讨论每个步骤。

B. Call Graph Extraction

The first step in MAMADROID is to extract the app's call graph. We do so by performing static analysis on the app's apk. 1 Specifically, we use a Java optimization and analysis framework, Soot [52], to extract call graphs and FlowDroid [6] to ensure contexts and flows are preserved. To better clarify the different steps involved in our system, we

B. Call Graph Extraction

在MAMADROID中，第一步是提取应用程序的调用图。我们通过对应用程序的apk进行静态分析来实现这一步骤。具体来说，我们使用了一个Java优化和分析框架Soot [52]来提取调用图，并使用FlowDroid [6]来确保上下文和流程得以保留。为了更清楚地说明我们系统中涉及的不同步骤，我们使用一个“运行示例”，使用一个伪装成内存加速器应用程序的真实恶意软件样本。具体来说，图2列出了从

employ a "running example," using a real-world malware sample. Specifically, Fig. 2 lists a class extracted from the decompiled apk of malware disguised as a memory booster app (with package name com.g.o.speed.memboost), which executes commands (rm, chmod, etc.) as root. 2 To ease presentation, we focus on the portion of the code executed in the try/catch block. The resulting call graph of the try/catch block is shown in Fig. 3. Note that, for simplicity, we omit calls for object initialization, return types and parameters, as well as implicit calls in a method. Additional calls that are invoked when getShell(true) is called are not shown, except for the add() method that is directly called by the program code, as shown in Fig. 2.

反编译的恶意软件apk中提取出的一个类（包名为com.g.o.speed.memboost），该类作为root用户执行命令（rm，chmod等）。为了简化展示，我们重点关注在try/catch块中执行的代码部分。try/catch块的结果调用图显示在图3中。请注意，为了简单起见，我们省略了对象初始化、返回类型和参数以及方法中的隐式调用。除了直接由程序代码调用的add()方法（如图2所示），在调用getShell(true)时触发的其他调用未显示出来。

C. Sequence Extraction

Next, we extract the sequences of API calls from the call graph. Since MAMADROID uses static analysis, the graph obtained from Soot represents the sequence of functions that are potentially called by the program. However, each execution of the app could take a specific branch of the graph and only execute a subset of the calls. For instance, when running the code in Fig. 2 multiple times, the Execute method could be followed by different calls, e.g., getShell() in the try block only or getShell() and then getMessage() in the catch block. In this phase, MAMADROID operates as follows. First, it identifies a set of entry nodes in the call graph, i.e., nodes with no incoming edges (for example, the Execute method in the snippet from Fig. 2 is the entry node if there is no incoming edge from any other call in the app). Then, it

C. Sequence Extraction

接下来，我们从调用图中提取API调用的序列。由于MAMADROID使用静态分析，从Soot中获取的图表示程序可能调用的函数的序列。然而，每次运行应用程序时，可能会选择图的特定分支并仅执行一部分调用。例如，多次运行图2中的代码时，Execute方法可能会后面跟随不同的调用，例如try块中只有getShell()，或者先是getShell()，然后是catch块中的getMessage()。

在这个阶段，MAMADROID的操作如下。首先，它识别调用图中的一组入口节点，即没有入边的节点（例如，如果应用程序中没有其他调用指向图2片段中的Execute方法，则该方法就是入口节点）。然后，它枚举可以从每个入口节点到达的路径。在此阶段识别的所有路径集合构成了将用于构建马尔可夫链行为模型和提取特征的API调用序列（详见第II-D节）。

对家族/包进行抽象化。与其分析原始的API调用，我们设计了MAMADROID以在较高的层

enumerates the (Return types and parameters are omitted to ease presentation). paths reachable from each entry node. The sets of all paths identified during this phase constitutes the sequences of API calls which will be used to build a Markov chain behavioral model and to extract features (see Section II-D). Abstracting Calls to Families/Packages. Rather than analyzing raw API calls, we build MAMADROID to work at a higher level, and operate in one of two modes by abstracting each call to either its package or family. This allows the system to be resilient to API changes and achieve scalability. In fact, our experiments, presented in Section III, show that, from a dataset of 44K apps, we extract more than 10 million unique API calls, which would result in a very large number of nodes, with the corresponding graphs (and feature vectors) being quite sparse. Since as we will see the number of features used by MAMADROID is the square of the number of nodes, having more than 10 million nodes would result in an impractical computational cost. When operating in package mode, we abstract an API call to its package name using the list of Android packages 3 , which as of API level 24 (the current version as of September 2016) includes 243 packages, as well as 95 from the Google API. 4 Moreover, we abstract developer-defined packages (e.g., com.stericson.roottools) as well as obfuscated ones (e.g. com.fa.a.b.d), respectively, as self-defined and obfuscated. Note that we label an API call's package as obfuscated if we cannot tell what its class implements, extends, or inherits, due to identifier mangling [47]. When operating in family mode, we abstract to nine possible families, i.e., android, google, java, javax, xml, apache, junit, json, dom, which correspond to the android., *com.google.*, java., *javax.*,

次上运行，并通过将每个调用抽象为其所属的包或家族中的一个来进行操作。这样做可以使系统对API的变化具有弹性并能够实现可扩展性。事实上，我们在第III节中的实验表明，从一个含有44K个应用程序的数据集中，我们提取了超过1000万个独特的API调用，这将导致非常多的节点，相应的图（和特征向量）将非常稀疏。由于正如我们将在后面看到的，MAMADROID使用的特征数量是节点数量的平方，拥有超过1000万个节点将导致计算成本过高。

在包模式下运行时，我们使用Android软件包列表将API调用抽象为其包名。该列表包括243个软件包（截至2016年9月的API级别24的版本），以及来自Google API的95个软件包。此外，我们将开发者定义的软件包（例如com.stericson.roottools）和混淆的软件包（例如com.fa.a.b.d）分别抽象为自定义和混淆。请注意，如果由于标识符混淆，我们无法确定API调用的类实现、扩展或继承的内容，我们将标记API调用的包为混淆。在家族模式下运行时，我们将API调用抽象为九个可能的家族，即android、google、java、javax、xml、apache、junit、json和dom，对应于android.、*com.google.*、java.、*javax.*、org.xml.、*org.apache.*、junit.、*org.json*和*org.w3c.dom.*软件包。同样，来自开发者定义和混淆的软件包的API调用将被抽象为分别标记为自定义和混淆的家族。总体而言，有340个（243+95+2）可能的软件包和11个（9+2）家族。在图4中，我们展示了从图3的调用图中获得的API调用序列。我们还在方括号中报告了调用所被抽象为的家族和包。

org.xml., *org.apache.*, junit., *org.json, and org.w3c.dom.* packages. Again, API calls from developerdefined and obfuscated packages are abstracted to families labeled as self-defined and obfuscated, respectively. Overall, there are 340 (243+95+2) possible packages and 11 (9+2) families. In Fig. 4, we show the sequence of API calls obtained from the call graph in Fig. 3. We also report, in square brackets, the family and the package to which the call is abstracted.

D. Markov-chain Based Modeling

Next, MAMADROID builds feature vectors, used for classification, based on the Markov chains representing the sequences of extracted API calls for an app. Before discussing this in detail, we review the basic concepts of Markov chains. Markov chains are memoryless models where the probability of transitioning from a state to another only depends on the current state [39]. Markov chains are often represented as a set of nodes, each corresponding to a different state, and a set of edges connecting one node to another labeled with the probability of that transition. The sum of all probabilities associated to all edges from any node (including, if present, an edge going back to the node itself) is exactly 1.The set of possible states of the Markov chain is denoted as S. If S j and S k are two connected states, P jk denotes the probability of transition from S j to S k . P jk is given by the number of occurrences (O jk) of state S k after state S j , divided by O ji for all states i in the chain, i.e., P jk = O jk i∈S Oji . Building the model. MAMADROID uses Markov chains to model app

D. Markov-chain Based Modeling

接下来，MAMADROID基于表示一个应用程序的提取的API调用序列的马尔可夫链建立特征向量，用于分类。在详细讨论此过程之前，我们回顾一下马尔可夫链的基本概念。

马尔可夫链是一种无记忆模型，其状态的转移概率只依赖于当前状态[39]。马尔可夫链通常表示为一组节点，每个节点对应不同的状态，以及一组边连接一个节点到另一个节点，边上标有该转移的概率值。从任何节点开始，所有边上的概率之和（包括可能存在的回到该节点的边）等于1。马尔可夫链的所有可能状态集合用S表示。如果S j 和S k 是两个连接的状态，则P jk 表示从S j 到S k 的转移概率。P jk 的计算基于状态 S j 后面出现状态 S k 的次数（O jk ），除以链中所有状态 i 的出现次数（O ji ），即 P jk = O jk i∈S Oji。

模型构建。MAMADROID使用马尔可夫链来建模应用程序行为，通过评估每个调用之间的转移。具体来说，对于每个应用程序，MAMADROID以该应用程序的抽象API调用序列作为输入，根据所选择的操作模式（即包或者家族）构建马尔可夫链，其中每个包/家族是一个状态，转移表示从一个状态到另一个状态的概率。对于每个马尔可夫链，状态 S 0 是入口点，其他调用按顺序执行。作为示例，图5展示了使用包和家族构建的两个马尔可夫链，分别基于图4中的序列。

behavior, by evaluating every transition between calls. More specifically, for each app, MAMADROID takes as input the sequence of abstracted API calls of that app -i.e., packages or families, depending on the selected mode of operation -and builds a Markov chain where each package/family is a state and the transitions represent the probability of moving from one state to another. For each Markov chain, state S 0 is the entry point from which other calls are made in a sequence. As an example, Fig. 5 illustrates the two Markov chains built using packages and families, respectively, from the sequences reported in Fig. 4. We argue that considering single transitions is more robust against attempts to evade detection by inserting useless API calls in order to deceive signature-based systems (see Section VI). In fact, MAMADROID considers all possible callsi.e., all the branches originating from a node -in the Markov chain, so adding calls would not significantly change the probabilities of transitions between nodes (specifically, families or packages, depending on the operational mode) for each app. Feature Extraction. Next, we use the probabilities of transitioning from one state (abstracted call) to another in the Markov chain as the feature vector of each app. States that are not present in a chain are represented as 0 in the feature vector. Also note that the vector derived from the Markov chain depends on the operational mode of MAMADROID. With families, there are 11 possible states, thus 121 possible transitions in each chain, while, when abstracting to packages, there are 340 states and 115,600 possible transitions. We also apply Principal Component Analysis (PCA) [32], which performs feature selection by transforming the feature space into a new space made of components that are a linear combination of the original

我们认为考虑单独的转移对抗检测更加强韧，因为它可以防止插入无用的API调用以欺骗基于签名的系统（见第VI节）。实际上，MAMADROID考虑马尔可夫链中的所有可能调用，即所有从一个节点发出的分支，因此添加调用不会显著改变每个应用程序间的转移概率（特别是对于每个应用程序的家族或包的操作模式）。

特征提取。接下来，我们使用马尔可夫链中从一个状态（抽象调用）转移到另一个状态的概率作为每个应用程序的特征向量。在链中不存在的状态在特征向量中表示为0。同时注意，马尔可夫链导出的向量取决于MAMADROID的操作模式。使用家族时，有11个可能的状态，因此每个链中有121个可能的转移；而当使用包时，有340个状态和115,600个可能的转移。

我们还应用主成分分析（PCA）[32]进行特征选择，通过将特征空间转换为原始特征的线性组合构成的新空间。前几个主成分包含尽可能多的方差（即信息量）。方差表示为原始特征空间总信息量的百分比。我们应用PCA来选择主要的成分，因为PCA将特征空间转换为一个较小的空间，在这个空间中，尽可能少的成分表示了方差，从而大大降低了计算和内存复杂度。此外，使用PCA还可以通过从特征空间中排除误导分类器性能的特征（即使分类器表现更差的特征），提高分类的准确性。

features. The first components contain as much variance (i.e., amount of information) as possible. The variance is given as percentage of the total amount of information of the original feature space. We apply PCA to the feature set in order to select the principal components, as PCA transforms the feature space into a smaller one where the variance is represented with as few components as possible, thus considerably reducing computation/memory complexity. Furthermore, the use of PCA could also improve the accuracy of the classification, by taking misleading features out of the feature space, i.e., those that make the classifier perform worse.

E. Classification

The last step is to perform classification, i.e., labeling apps as either benign or malware. To this end, we test MAMADROID using different classification algorithms: Random Forests [9], 1-Nearest Neighbor (1-NN) [22], 3-Nearest Neighbor (3-NN) [22], and Support Vector Machines (SVM) [29]. Each model is trained using the feature vector obtained from the apps in a training sample. Results are presented and discussed in Section IV, and have been validated by using 10-fold cross validation. Also note that, due to the different number of features used in family/package modes, we use two distinct configurations for the Random Forests algorithm. Specifically, when abstracting to families, we use 51 trees with maximum depth 8, while, with packages, we use 101 trees of maximum depth 64. To tune Random Forests we followed the methodology applied in [7].

E. Classification

最后一步是进行分类，即将应用程序标记为良性或恶意。为此，我们使用不同的分类算法对MAMADROID进行测试：随机森林[9]，1-最近邻算法（1-NN）[22]，3-最近邻算法（3-NN）[22] 和支持向量机（SVM）[29]。每个模型都使用训练样本中的应用程序获得的特征向量进行训练。结果将在第四节中展示和讨论，并且通过使用10折交叉验证进行验证。另请注意，由于在家族/包模式中使用了不同数量的特征，我们为随机森林算法使用了两种不同的配置。具体而言，在抽象到家族时，我们使用了51棵最大深度为8的树，而在抽象到包时，我们使用了101棵最大深度为64的树。为调整随机森林算法，我们遵循了[7]中的方法论。

III. DATASETS

In this section, we introduce the datasets used in the evaluation of MAMADROID (presented later in Section IV), 6 as of March 7, 2016, using the googleplayapi tool. 7 Due to errors encountered while downloading some apps, we have actually obtained 2,843 out of 2,900 apps. Note that 275 of these belong to more than one category, therefore, the newbenign dataset ultimately includes 2,568 unique apps. Android Malware Samples. The set of malware samples includes apps that were used to test DREBIN [5], dating back to October 2010 -August 2012 (5,560), which we denote as drebin, as well as more recent ones that have been uploaded on the VirusShare8 site over the years. Specifically, we gather from VirusShare, respectively, 6,228, 15,417, 5,314, and 2,974 samples from 2013, 2014, 2015, and 2016. We consider each of these datasets separately for our analysis. API Calls and Call Graphs. For each app in our datasets, we extract the list of API calls, using Androguard9 , since, as explained in Section IV-E, these constitute the features used by DROIDAPIMINER [2], against which we compare our system. Due to Androguard failing to decompress some of the apks, bad CRC-32 redundancy checks, and errors during unpacking, we are not able to extract the API calls for all the samples, but only for 40,923 (8,402 benign, 34,521 malware) out of the 43,940 apps (8,447 benign, 35,493 malware) in our datasets. Also, to extract the call graph of each apk, we use Soot. Note that for some of the larger apks, Soot requires a non-negligible amount of memory to extract the call graph, so we allocate 16GB of RAM to the Java VM heap space. We find that for 2,472 (364 benign + 2,108

III. DATASETS

III. 数据集

在本节中，我们介绍了在评估MAMADROID时使用的数据集（在第四节中介绍），截至2016年3月7日，使用了googleplayapi工具。由于在下载某些应用程序时遇到了错误，我们实际上只得到了2,843个中的2,900个应用程序。需要注意的是，其中的275个属于多个类别，因此新的良性数据集最终包括2,568个唯一的应用程序。

Android恶意样本。恶意样本集合包括用于测试DREBIN [5]的应用程序，时间跨度从2010年10月至2012年8月（共5,560个），我们将其称为drebin，以及随着时间流逝上传到VirusShare8网站上的最新样本。具体来说，我们从VirusShare分别收集了来自2013年、2014年、2015年和2016年的6,228、15,417、5,314和2,974个样本。我们将每个数据集分别进行分析。

API调用和调用图。对于我们数据集中的每个应用程序，在使用DROIDAPIMINER对比分析时，我们使用Androguard9提取API调用列表，因为如第IV-E节所解释的那样，这些构成了DROIDAPIMINER使用的特征。由于Androguard在解压缩某些apk文件时出错、CRC-32冗余校验失败和解包过程中出现错误，我们无法提取所有样本的API调用，而只能针对其中的40,923个样本（包括8,402个良性样本和34,521个恶意样本）进行提取，而数据集中共有43,940个应用程序（包括8,447个良性样本和35,493个恶意样本）。此外，为了提取每个apk的调用图，我们使用了Soot工具。需要注意的是，对于一些较大的apk文件，Soot需要分配相当大的内存来提取调用图，因此我们将16GB的内存分配给Java虚拟机堆空间。我们发现，在2,472个样本（其中包括364个良性样本和2,108个恶意样本）中，由于Soot无法完成提取（无法应用jb阶段以及报告在打开某些zip文件（即apk文件）时出错），因此我们在评估中排除了这些应用程序，并在第V-C节中进一步讨论了这个

malware) samples, Soot is not able to complete the extraction due to it failing to apply the jb phase as well as reporting an error in opening some zip files (i.e., the apk). The jb phase is used by Soot to transform Java bytecode into jimple intermediate representation (the primary IR of Soot) for optimization purposes. Therefore, we exclude these apps in our evaluation and discuss this limitation further in Section V-C. In Table I, we provide a summary of our seven datasets, reporting the total number of samples per dataset, as well as those for which we are able to extract the API calls (second-to-last column) and the call graphs (last column). Characterization of the Datasets. Aiming to shed light on the evolution of API calls in Android apps, we also performed some measurements over our datasets. In Fig. 6, we plot the Cumulative Distribution Function (CDF) of the number of unique API calls in the apps in different datasets, highlighting that newer apps, both benign and malicious, are using more API calls overall than older apps. This indicates that as time goes by, Android apps become more complex. When looking at the fraction of API calls belonging to specific families, we discover some interesting aspects of Android apps developed in different years. In particular, we notice that API calls to the android family become less prominent as time passes (Fig. 7(a)), both in benign and malicious datasets, while google calls become more common in newer apps (Fig. 7(b)). In general, we conclude that benign and malicious apps show the same evolutionary trends over the years. Malware, however, appears to reach the same characteristics (in terms of level of complexity and fraction of API calls from certain families) as legitimate apps with a few years of delay. Principal Component Analysis. Finally, we apply PCA to select the two most important

限制。在表I中，我们提供了我们的七个数据集的摘要，报告了每个数据集的样本总数，以及我们能够提取API调用（倒数第二列）和调用图（最后一列）的样本数。

数据集的特征化。为了揭示Android应用程序中API调用的演变情况，我们还对我们的数据集进行了一些测量。在图6中，我们绘制了不同数据集中应用程序的唯一API调用数量的累积分布函数（CDF），突显出较新的应用程序（包括良性和恶意）总体上使用更多的API调用比较旧的应用程序。这表明随着时间的推移，Android应用程序变得更加复杂。当查看特定家族的API调用所占比例时，我们发现了不同年份开发的Android应用程序的一些有趣特点。特别是，我们注意到随着时间的推移，良性和恶意数据集中对android家族的API调用变得不那么突出（图7(a)），而对于新的应用程序，对google的调用变得更为常见（图7(b)）。

总体而言，我们得出结论，良性和恶意应用程序在多年的发展过程中显示出相同的演变趋势。然而，恶意软件似乎会在几年后达到与合法应用程序相同的特征（在复杂性水平和特定家族API调用的比例方面）。

主成分分析。最后，我们运用PCA来选择最重要的两个主成分。我们绘制并对比了良性样本（图8(a)）和恶意样本（图8(b)）在两个主成分上的位置。由于PCA将特征合并为主成分，它最大化了样本在这些主成分上的分布方差，因此绘制样本在主成分空间上的位置表明，良性应用程序往往位于不同的区域，具体取决于数据集，而恶意样本则占据相似的区域，但密度不同。这些差异突出了良性和恶意样本之间的不同行为，机器学习算法用于分类时也应该能够发现这些差异。

PCA components. We plot and compare the positions of the two components for benign (Fig. 8(a)) and malicious samples (Fig. 8(b)). As PCA combines the features into components, it maximizes the variance of the distribution of samples in these components, thus, plotting the positions of the samples in the components shows that benign apps tend to be located in different areas of the components space, depending on the dataset, while malware samples occupy similar areas but with different densities. These differences highlight a different behavior between benign and malicious samples, and these differences should also be found by the machine learning algorithms used for classification.

IV. EVALUATION

IV. EVALUATION

We now present a detailed experimental evaluation of MAMADROID. Using the datasets summarized in Table I, we perform four sets of experiments: (1) we analyze the accuracy of MAMADROID's classification on benign and malicious samples developed around the same time; (2) we evaluate its robustness to the evolution of malware as well as of the Android framework by using older datasets for training and newer ones for testing (and vice-versa); (3) we measure MAMADROID's runtime performance to assess its scalability; and, finally, (4) we compare against DROIDAPIMINER [2], a malware detection system that relies on the frequency of API calls.

四、评估

我们现在对MAMADROID进行详细的实验评估。使用在表一中总结的数据集，我们执行了四组实验：(1)我们分析了MAMADROID在同时开发的良性样本和恶意样本上的分类准确性；(2)我们通过使用较旧的数据集进行训练和使用较新的数据集进行测试（或反之亦然），评估了其对恶意软件和Android框架的演化的鲁棒性；(3)我们测量了MAMADROID的运行时性能，评估其可扩展性；最后，(4)我们与依赖于API调用频率的恶意软件检测系统DROIDAPIMINER [2]进行了比较。

A. Preliminaries

A. Preliminaries

When implementing MAMADROID in family mode, we exclude the json and

在实现MAMADROID的族群模式下，我们排除了 json 和 dom 族群，因为它们在我们的

dom families because they are almost never used across all our datasets, and junit, which is primarily used for testing. In package mode, to avoid mislabeling when self-defined APIs have "android" in the name, we split the android package into its two classes, i.e., android.R and android.Manifest. Therefore, in family mode, there are 8 possible states, thus 64 features, whereas, in package mode, we have 341 states and 116,281 features (cf. Section II-D). As discussed in Section II-E, we use four different machine learning algorithms for classification -namely, Random Forests [9], 1-NN [22], 3-NN [22], and SVM [29]. Since both accuracy and speed are worse with SVM than with the other three algorithms, we omit results obtained with SVM. To assess the accuracy of the classification, we use the standard F-measure metric, i.e.: F = 2 • precision • recall precision + recall where precision = TP/(TP+FP) and recall = TP/(TP+FN). TP denotes the number of samples correctly classified as malicious, while FP an FN indicate, respectively, the number of samples mistakenly identified as malicious and benign. Finally, note that all our experiments perform 10-fold cross validation using at least one malicious and one benign dataset from Table I. In other words, after merging the datasets, the resulting set is shuffled and divided into ten equal-size random subsets. Classification is then performed ten times using nine subsets for training and one for testing, and results are averaged out over the ten experiments.

所有数据集中几乎没有被使用过，以及用于测试的 junit。在包模式下，为了避免当自定义的 API 中带有 "android" 时的错误标记，我们将 android 包分为其两个类，即 android.R 和 android.Manifest。因此，在族群模式下，有8个可能的状态，即64个特征，而在包模式下，我们有341个状态和116,281个特征（参见第II-D节）。

正如第II-E节中讨论的，我们使用了四种不同的机器学习算法进行分类，分别是随机森林 [9]、1-NN [22]、3-NN [22] 和 SVM [29]。由于使用 SVM 的准确性和速度都不如其他三种算法，我们省略了使用 SVM 获得的结果。为了评估分类的准确性，我们使用标准的 F-度量指标，即： F = 2 • precision • recall precision + recall 其中 precision = TP / (TP + FP)，recall = TP / (TP + FN)。TP 表示正确分类为恶意的样本数，而 FP 和 FN 分别表示被错误地识别为恶意和良性的样本数。

最后，请注意，我们的所有实验都使用了至少来自表I的一个恶意数据集和一个良性数据集进行10倍交叉验证。换句话说，在合并数据集后，结果集被随机打乱并分为十个相等大小的随机子集。然后进行十次分类，使用九个子集进行训练，一个进行测试，然后将结果在十个实验中进行平均。

B. Detection Performance

B. Detection Performance

We start our evaluation by measuring how well MA-MADROID detects malware

我们首先通过在开发时间相近的样本上进行训练和测试来评估MAMADROID对恶意软件的

by training and testing using samples that are developed around the same time. To this end, we perform 10-fold cross validations on the combined dataset composed of a benign set and a malicious one. Table II provides an overview of the detection results achieved by MAMADROID on each combined dataset, in the two modes of operation, both with PCA features and without. The reported F-measure, precision, and recall scores are the ones obtained with Random Forest, which generally performs better than 1-NN and 3-NN. Family mode. In Fig. 9, we report the F-measure when operating in family mode for Random Forests, 1-NN and 3-NN. The F-measure is always at least 88% with Random Forests, and, when tested on the 2014 (malicious) dataset, it reaches 98%. With some datasets, MAMADROID performs slightly better than with others. For instance, with the 2014 malware dataset, we obtain an F-measure of 92% when using the oldbenign dataset and 98% with newbenign. In general, lower F-measures are due to increased false positives since recall is always above 91%, while precision might be lower, also due to the fact that malware datasets are larger than the benign sets. We believe that this follows the evolutionary trend discussed in Section III: while both benign and malicious apps become more complex as time passes, when a new benign app is developed, it is still possible to use old classes or re-use code from previous versions and this might cause them to be more similar to old malware samples. This would result in false positives by MAMADROID. In general, MAMADROID performs better when the different characteristics of malicious and benign training and test sets are more predominant, which corresponds to datasets occupying different positions of

检测效果。为此，我们对由良性集和恶意集组成的合并数据集进行了10折交叉验证。表 II 提供了MAMADROID在每个合并数据集上的检测结果概览，包括两种操作模式下的结果，分别使用了PCA特征和未使用PCA特征。报告的F-measure、精确度和召回率分数是使用随机森林得到的，随机森林通常比最近邻1-NN和3-NN表现更好。

家族模式。在图9中，我们报告了以家族模式运行随机森林、最近邻1-NN和3-NN时的F-measure。随机森林的F-measure始终至少为88%，当在2014（恶意）数据集上进行测试时，它达到了98%。MAMADROID在某些数据集上的表现略好于其他数据集。例如，在2014恶意软件数据集上，我们使用oldbenign数据集时获得了92%的F-measure，使用newbenign数据集时获得了98%的F-measure。通常，较低的F-measure是由于增加的误报率，因为召回率始终在91%以上，而精确度可能较低，这也是由于恶意软件数据集大于良性集的原因。我们认为这符合第三节中讨论的演化趋势：随着时间的推移，良性和恶意应用程序都变得更加复杂，但当开发一个新的良性应用程序时，仍然可以使用旧的类或重用以前版本的代码，这可能导致它们与旧的恶意软件样本更相似。这将导致MAMADROID产生误报。总体而言，在训练和测试集的不同特征更明显的情况下，MAMADROID的表现更好，这对应于数据集在特征空间不同位置。

包模式。当MAMADROID以包模式运行时，分类性能提高，使用随机森林时，F-measure范围从2016和newbenign的92%到2014和newbenign的99%。图10报告了使用随机森林、最近邻1-NN和3-NN（包模式）进行的10折交叉验证实验的F-measure。在这种情况下，随机森林通常提供更好的结果。

在某些数据集上，两种操作模式之间的性能差异更为明显：对于drebin和oldbenign，并使用随机森林，我们在包模式下获得了96%的F-measure，而在家族模式下为88%。这些差异是由于包模式下的误报率较低所引起的。召回率保持较高，整体上形成更平衡的系统。总体而言，将抽象到包级别而不是家族级别提供更好的结果，因为增加的粒度能够识别出更多

the feature space. Package mode. When MAMADROID runs in package mode, the classification performance improves, ranging from 92% F-measure with 2016 and newbenign to 99% with 2014 and newbenign, using Random Forests. Fig. 10 reports the Fmeasure of the 10-fold cross validation experiments using Random Forests, 1-NN, and 3-NN (in package mode). The former generally provide better results also in this case. With some datasets, the difference in performance between the two modes of operation is more noticeable: with drebin and oldbenign, and using Random Forests, we get 96% Fmeasure in package mode compared to 88% in family mode. These differences are caused by a lower number of false positives in package mode. Recall remains high, resulting in a more balanced system overall. In general, abstracting to packages rather than families provides better results as the increased granularity enables identifying more differences between benign and malicious apps. On the other hand, however, this likely reduces the efficiency of the system, as many of the states deriving from the abstraction are used a only few times. The differences in time performance between the two modes are analyzed in details in Section IV-F. Using PCA. As discussed in Section II-D, PCA transforms large feature spaces into smaller ones, thus it can be useful to significantly reduce computation and, above all, memory complexities of the classification task. When operating in package mode, PCA is particularly beneficial, since MAMADROID originally has to operate over 116,281 features. Therefore, we compare results obtained using PCA by fixing the number of components to 10 and checking the quantity of variance included in them. In package mode, we observe that only 67% of the variance is taken into account by

的良性和恶意应用程序之间的差异。然而，另一方面，这可能会减少系统的效率，因为从抽象中产生的状态只使用了几次。两种操作模式之间的时间性能差异在第四节的IV-F中进行了详细分析。

使用PCA。正如在第二节D节中讨论的，PCA将大的特征空间转换为较小的特征空间，因此可以显著减少分类任务的计算复杂性和内存复杂性。当以包模式运行时，PCA特别有益，因为MAMADROID原本要处理116,281个特征。因此，我们通过将组件数量固定为10并检查其中包含的方差量来比较使用PCA的结果。在包模式下，我们观察到只有最重要的10个PCA组件考虑了67%的方差，而在家族模式下，至少有91%的方差通过10个PCA组件（使用完整的特征集）得到了考虑。我们注意到，较低的F-measure是由于精确度和召回率均匀下降所引起的。

the 10 most important PCA components, whereas, in family mode, at least 91% of the variance is included by the 10 PCA Components. using the full feature set. We note that lower F-measures are caused by a uniform decrease in both precision and recall.

C. Detection Over Time

As Android evolves over the years, so do the characteristics of both benign and malicious apps. Such evolution must be taken into account when evaluating Android malware detection systems, since their accuracy might significantly be affected as newer APIs are released and/or as malicious developers modify their strategies in order to avoid detection. Evaluating this aspect constitutes one of our research questions, and one of the reasons why our datasets span across multiple years (2010-2016). As discussed in Section II-B, MAMADROID relies on the sequence of API calls extracted from the call graphs and abstracted at either the package or the family level. Therefore, it is less susceptible to changes in the Android API than other classification systems such as DROIDAPIMINER [2] and DREBIN [5]. Since these rely on the use, or the frequency, of certain API calls to classify malware vs benign samples, they need to be retrained following new API releases. On the contrary, retraining is not needed as often with MAMADROID, since families and packages represent more abstract functionalities that change less over time. Consider, for instance, the android.os.health package: released with API level 24, it contains a set of classes helping developers track and monitor system resources. 10 Classification systems built before this release -as in the case of DROIDAPIMINER [2] (released in 2013, when Android API was up to level 20) -need to be retrained if this package is more frequently used by malicious apps than benign apps, while MAMADROID only needs to add a new state to its Markov chain when operating in package mode, while no additional state is required when operating in family mode. To verify this hypothesis, we test MAMADROID using older samples as training sets and newer ones as test sets. Fig. 11 reports the F-measure of the classification in this setting, with MAMADROID operating in family mode. The x-axis reports the difference in years between training and test data. We obtain 86% F-measure when we classify apps one year older than the samples on which we train. Classification is still relatively accurate, at 75%, even after two years. Then, from Fig. 12, we observe that the F-measure does not significantly change when operating in package mode. Both modes of operations are affected by one particular condition, already discussed in Section III: in our models, benign datasets seem to "anticipate" malicious ones by 1-2 years in the way they use certain API calls. As a result, we notice a drop in accuracy when 10 https://developer.android.com/reference/android/os/health/package-summary. html More specifically, we observe that MAMADROID correctly detects benign apps, while it starts missing true positives and increasing false negatives -i.e., achieving lower recall. We also set to verify whether older malware samples can still

be detected by the system-if not, this would obviously become vulnerable to older (and possibly popular) attacks. Therefore, we also perform the "opposite" experiment, i.e., training MAMADROID with newer datasets, and checking whether it is able to detect malware developed years before. Specifically, Fig. 13 and 14 report results when training MAMADROID with samples from a given year, and testing it with others that are up to 4 years older: MAMADROID retains similar F-measure scores over the years. Specifically, in family mode, it varies from 93% to 96%, whereas, in package mode, from 95% to 97% with the oldest samples.

C. Detection Over Time

随着Android系统的演进，无论是良性应用还是恶意应用的特征都在不断变化。在评估Android恶意软件检测系统时，必须考虑到这种演进，因为随着新的API发布或者恶意开发者修改策略以避免被检测，系统的准确性可能会受到显著影响。评估这一方面是我们的研究问题之一，也是为什么我们的数据集跨越多年（2010年至2016年）的原因之一。

如第II-B节中所讨论的，MAMADROID依赖于从调用图中提取的API调用序列，这些序列在包或者家族级别上进行了抽象。因此，相比于其他分类系统（如DROIDAPIMINER [2]和DREBIN [5]），MAMADROID对Android API的变化不太敏感。由于这些系统依赖于特定的API调用的使用或频率来对恶意样本与良性样本进行分类，它们需要在新的API发布后重新训练。相反，MAMADROID不需要频繁重新训练，因为家族和包更加抽象的功能在时间上变化较少。例如，考虑android.os.health包：它在API级别24发布，包含一组帮助开发者跟踪和监控系统资源的类。在此发布之前构建的10个分类系统，如DROIDAPIMINER [2]（在2013年发布，当时Android API的级别为20），如果这个包由恶意应用更频繁地使用而不是良性应用，就需要重新训练，而MAMADROID只需要在包模式下向它的马尔可夫链中添加一个新的状态，而在家族模式下则不需要添加额外的状态。

为了验证这一假设，我们使用较早样本作为训练集，较新样本作为测试集来测试MAMADROID。图11报告了MAMADROID在家族模式下进行分类的F-measure。横轴表示训练数据与测试数据之间的年份差异。当我们对比训练样本比测试样本早一年的应用进行分类时，我们获得了86%的F-measure。即使经过两年，分类仍然相对准确，为75%。然后，从图12中可以看出，在包模式下，F-measure并没有显著变化。两种操作模式都受到已在第III节中讨论的一个特定条件的影响：在我们的模型中，良性数据集似乎“提前”1-2年使用某些API调用方式来预测出恶意样本。因此，当这些API调用在良性应用中发生改变时，准确性会下降，MAMADROID无法正确检测到这些恶意应用，导致漏报率增加。

我们还试图验证系统能否检测到较旧的恶意样本-如果不能，那么它显然就会对较旧（可能也较流行）的攻击造成漏洞。因此，我们还进行了“相反”的实验，即使用更新的数据集来训练MAMADROID，并检查它是否能够检测多年前开发的恶意软件。具体来说，图13和图14报告了在给定年份的样本上训练MAMADROID，并使用比其早4年的其他样本进行测试时的结果：在家族模式下，它的F-measure得分变化不大，从93%到96%；而在包模式下，从95%到97%，即使是对于最旧的样本。

D. Case Studies of False Positives and Negatives

The experiment analysis presented above show that MA-MADROID detects Android malware with high accuracy. As in any detection system, however, the system makes a small number of incorrect classifications, incurring some false positives and false negatives. Next, we discuss a few case studies aiming to better understand these misclassifications. We focus on the experiments with newer datasets, i.e., 2016 and newbenign. False Positives. We analyze the manifest of the 164 apps mistakenly detected as malware by MAMADROID, finding that most of them use "dangerous" permissions [4]. In particular, 67% of the apps write to external storage, 32% read the phone state, and 21% access the device's fine location. We further analyzed apps (5%) that use the READ_SMS and SEND_SMS permissions, i.e., even though they are not SMS-related apps, they can read and send SMSs as part of the services they provide to users. In particular, a "in case of emergency" app is able to send messages to several contacts from its database (possibly added by the user), which is a typical behavior of Android malware in our dataset, ultimately leading MAMADROID to flag it as malicious. False Negatives. We also check the 114 malware samples missed by MAMADROID when operating in family mode, using VirusTotal. 11 We find that 18% of the false negatives are actually not classified as malware by any of the antivirus engines used by VirusTotal, suggesting that these are actually

D. Case Studies of False Positives and Negatives

上述的实验分析显示，MA-MADROID能够高准确度地检测Android恶意软件。然而，像任何检测系统一样，该系统会产生一小部分错误分类，导致一些误报和漏报。接下来，我们将讨论几个案例研究，旨在更好地理解这些错误分类。我们重点关注对较新数据集（即2016年和newbenign）进行的实验。

误报案例。我们分析了被MAMADROID错误检测为恶意软件的164个应用的清单，发现其中大部分使用了“危险”权限[4]。特别是，67%的应用会写入外部存储，32%读取手机状态，21%访问设备的精确定位。我们进一步分析了使用READ_SMS和SEND_SMS权限的应用（占5%），即使它们不是短信相关应用，它们也可以以提供给用户的服务的一部分读取和发送短信。特别是，一个“紧急情况下”的应用可以从数据库中向多个联系人发送短信（可能由用户添加），这是我们数据集中Android恶意软件的典型行为，最终导致MAMADROID将其标记为恶意。

漏报案例。我们还使用VirusTotal对MAMADROID在家族模式下错过的114个恶意软件样本进行了检查 [11]。我们发现，18%的漏报实际上没有被VirusTotal使用的任何杀毒引擎分类为恶意软件，这表明这些实际上是错误包含在VirusShare数据集中的合法应用。45%的MAMADROID漏报是广告软件，通常是重新打包的应用程序，其中广告库已被替换为第三方库，为开发者带来经济利益。由于它们没有执行明显恶意的活动，MAMADROID无法将其识别为恶意软件。最后，我们发现MAMADROID报告的16%的漏报是样本发送短信或拨打高价服务电话。我们还对抽象为包的漏报样本进行了类似的分析（74个样本），结果类似：广告软件样本稍多（53%），潜在的良性应用样本比例相似

legitimate apps mistakenly included in the VirusShare dataset. 45% of MAMADROID's false negatives are adware, typically, repackaged apps in which the advertisement library has been substituted with a third-party one, which creates a monetary profit for the developers. Since they are not performing any clearly malicious activity, MAMADROID is unable to identify them as malware. Finally, we find that 16% of the false negatives reported by MAMADROID are samples sending text messages or starting calls to premium services. We also do a similar analysis of false negatives when abstracting to packages (74 samples), with similar results: there a few more adware samples (53%), but similar percentages for potentially benign apps (15%) and samples sending SMSs or placing calls (11%). In conclusion, we find that MAMADROID's sporadic misclassifications are typically due to benign apps behaving similarly to malware, malware that do not perform clearlymalicious activities, or mistakes in the ground truth labeling.

（15%），以及发送短信或拨打电话的样本比例（11%）。

总结起来，我们发现MAMADROID的零星错误分类通常是由于良性应用与恶意软件类似的行为、恶意软件没有执行明显恶意活动，或者在基准标签中存在错误。

E. MAMADROID vs DROIDAPIMINER Part-1

We also compare the performance of MAMADROID to previous work using API features for Android malware classification. Specifically, we compare to DROIDAPIMINER [2], because: (i) it uses API calls and its parameters to perform classification; (ii) it reports high true positive rate (up to 97.8%) on almost 4K malware samples obtained from McAfee and GENOME [66], and 16K benign

E. MAMADROID vs DROIDAPIMINER

我们还将MAMADROID与使用API特征进行Android恶意软件分类的先前工作进行了性能比较。具体而言，我们与DROIDAPIMINER [2]进行了比较，原因如下：(i) DROIDAPIMINER使用API调用及其参数进行分类；(ii) 它在从McAfee和GENOME [66]获取的近4K个恶意软件样本和16K个良性样本上报告了高的真正阳性率（高达97.8%）；(iii) 该系统的源代码已由作者提供给我们。

在DROIDAPIMINER中，使用恶意软件样本请求频率高于良性应用程序的权限进行基准分

samples; and (iii) its source code has been made available to us by the authors. In DROIDAPIMINER, permissions that are requested more frequently by malware samples than by benign apps are used to perform a baseline classification. Since there are legitimate situations where a non-malicious app needs permissions tagged as dangerous, DROIDAPIMINER also applies frequency analysis on the list of API calls, specifically, using the 169 most frequent API calls in the malware samples (occurring at least 6% more in malware than benign samples) -leading to a reported 83% precision. Finally, data flow analysis is applied on the API calls that are frequent in both benign and malicious samples, but do not occur by at least, 6% more in the malware set. Using the top 60 parameters, the 169 most frequent calls change, and authors report a precision of 97.8%.

类。由于存在合法情况下非恶意应用程序需要标记为危险权限的情况，DROIDAPIMINER还对API调用列表进行频率分析，具体而言，使用在恶意软件样本中至少比良性样本多6%的神经元热门API调用列表（导致报告的精确度为83%）。最后，对在良性样本和恶意样本中频繁出现但在恶意软件集中至少不多于6%的API调用进行数据流分析。使用前60个参数，这169个最频繁的调用将发生变化，作者报告了精确度为97.8%。

E. MAMADROID vs DROIDAPIMINER Part-2

After obtaining DROIDAPIMINER's source code, as well as a list of packages used for feature refinement, we re-implement the system by modifying the code in order to reflect recent changes in Androguard (used by DROIDAPIMINER for API call extraction), extract the API calls for all apps in the datasets listed in Table I, and perform a frequency analysis on the calls. Androguard fails to extract calls for about 2% (1,017) of apps in our datasets as a result of bad CRC-32 redundancy checks and error in unpacking, thus DROIDAPIMINER is evaluated over the samples in the second-to-last column of Table I. We also

E. MAMADROID vs DROIDAPIMINER

获得DROIDAPIMINER的源代码以及用于特征细化的包列表后，我们通过修改代码来重新实现系统，以反映Androguard的最新变化（Androguard用于提取API调用），提取表I中所列数据集中所有应用程序的API调用，并对调用进行频率分析。由于Androguard在提取API调用时存在差错的CRC-32冗余检查和解包错误，导致我们数据集中约2%（1,017个）的应用程序无法提取调用，因此我们对Table I倒数第二列中的样本进行了对DROIDAPIMINER的评估。此外，我们还实现了分类功能（这是作者提供的代码中缺失的部分），使用k-NN（其中k=3）进行分类，因为根据论文，k-NN的效果最好。我们按照作者的实现方法，将数据集的2/3用于训练，1/3用于测试[2]。不同训练和测试集所得到的F-measure的总结结果见表III。

implement classification, which is missing from the code provided by the authors, using k-NN (with k=3) since it achieves the best results according to the paper. We use 2/3 of the dataset for training and 1/3 for testing as implemented by the authors [2]. A summary of the resulting F-measures obtained using different training and test sets is presented in Table III.

E. MAMADROID vs DROIDAPIMINER Part-3

We set up a number of experiments to thoroughly compare DROIDAPIMINER to MAMADROID. First, we set up three experiments in which we train DROIDAPIMINER using a dataset composed of oldbenign combined with one of the three oldest malware datasets each (drebin, 2013, and 2014), and testing on all malware datasets. With this configuration, the best result (with 2014 and oldbenign as training sets) amounts to 62% F-measure when tested on the same dataset. The F-measure drops to 33% and 39%, respectively, when tested on samples one year into the future and past. If we use the same configurations in MAMADROID, in package mode, we obtain up to 97% F-measure (using 2013 and oldbenign as training sets), dropping to 73% and 94%, respectively, one year into the future and into the past. For the datasets where DROIDAPIMINER achieves its best result (i.e., 2014 and oldbenign), MAMADROID achieves an F-measure of 95%, which drops to respectively, 78% and 93% one year into the future and the past. The F-measure is stable even two

E. MAMADROID vs DROIDAPIMINER

我们设置了一系列实验来全面比较MAMADROID和DROIDAPIMINER。首先，我们进行了三个实验，其中训练集由老版本的良性应用程序和三个最早的恶意应用程序数据集（drebin、2013和2014）组成，然后在所有恶意应用程序数据集上进行测试。在这种配置下，最佳结果（使用2014和老版本良性应用程序作为训练集）的F-measure为62%。当在将来一年和过去一年的样本上进行测试时，F-measure分别下降到33%和39%。如果我们在MAMADROID中使用相同的配置，以包模式运行，我们可以获得高达97%的F-measure（使用2013和老版本良性应用程序作为训练集），当分别在将来一年和过去一年的样本上进行测试时，F-measure分别下降到73%和94%。对于DROIDAPIMINER在其最佳结果（即2014和老版本良性应用程序）上实现的数据集，MAMADROID的F-measure为95%，当分别在将来一年和过去一年的样本上进行测试时，F-measure分别下降到78%和93%。即使是在将来两年和过去两年的情况下，F-measure也保持稳定，分别为75%和92%。

years into the future and the past at 75% and 92%, respectively.

E. MAMADROID vs DROIDAPIMINER Part-4

As a second set of experiments, we train DROIDAPIMINER using a dataset composed of newbenign combined with one of the three most recent malware datasets each (2014, 2015, and 2016). Again, we test DROIDAPIMINER on all malware datasets. The best result is obtained with the dataset (2014 and newbenign) used for both testing and training, yielding a Fmeasure of 92%, which drops to 67% and 75% one year into the future and past respectively. Likewise, we use the same datasets for MAMADROID, with the best results achieved on the same dataset as DROIDAPIMINER. In package mode, MAMADROID achieves an F-measure of 99%, which is maintained more than two years into the past, but drops to respectively, 85% and 81% one and two years into the future. As summarized in Table III, MAMADROID achieves significantly higher performance than DROIDAPIMINER in all but one experiment, with the F-measure being at least 75% even after two years into the future or the past when datasets from 2014 or later are used for training. Note that there is only one setting in which DROIDAPIMINER performs slightly better than MAMADROID: this occurs when the malicious training set is much older than the malicious test set. Specifically, MAMADROID presents low recall in this case: as discussed, MAMADROID's classification performs much better when

E. MAMADROID vs DROIDAPIMINER

作为第二组实验，我们使用由新的良性样本和最近三个恶意软件数据集之一（2014年、2015年和2016年）组成的数据集来训练DROIDAPIMINER。同样，我们在所有的恶意软件数据集上测试DROIDAPIMINER。在测试和训练中使用数据集（2014年和新的良性样本）时，最佳结果为92%的F值，但在未来一年和过去一年时，这一值分别下降至67%和75%。同样，我们使用相同的数据集来训练MAMADROID，并在与DROIDAPIMINER相同的数据集上取得了最佳的结果。在包模式下，MAMADROID的F值为99%，在过去两年中保持稳定，但在未来一年和未来两年分别下降至85%和81%。总结在表III中，当使用2014年或之后的数据集来进行训练时，MAMADROID在除了一个实验之外均表现出显著更高的性能，其F值在未来两年或过去两年至少为75%。值得注意的是，只有在恶意训练集比恶意测试集旧得多的情况下，DROIDAPIMINER表现略优于MAMADROID。具体而言，在这种情况下，MAMADROID的召回率较低：正如讨论过的，当训练集不超过测试集的两年时，MAMADROID的分类效果要好得多。

the training set is not more than two years older than the test set.

F. Runtime Performance Part-1

We envision MAMADROID to be integrated in offline detection systems, e.g., run by Google Play. Recall that MAMADROID consists of different phases, so in the following, we review the computational overhead incurred by each of them, aiming to assess the feasibility of real-world deployment. We run our experiments on a desktop equipped with an 40-core 2.30GHz CPU and 128GB of RAM, but only use one core and allocate 16GB of RAM for evaluation. MAMADROID's first step involves extracting the call graph from an apk and the complexity of this task varies significantly across apps. On average, it takes 9.2s±14 (min 0.02s, max 13m) to complete for samples in our malware sets. Benign apps usually yield larger call graphs, and the average time to extract them is 25.4s±63 (min 0.06s, max 18m) per app. Note that we do not include in our evaluation apps for which we could not successfully extract the call graph. Next, we measure the time needed to extract call sequences while abstracting to families or packages, depending on MAMADROID's mode of operation. In family mode, this phase completes in about 1.3s on average (and at most 11.0s) with both benign and malicious samples. Abstracting to packages takes slightly longer, due to the use of 341 packages in MAMADROID. On average, this extraction takes 1.67s±3.1 for malicious apps and 1.73s±3.2 for benign samples. As it can be seen, the call sequence extraction in package mode

F. Runtime Performance

我们设想MAMADROID将被整合到离线检测系统中，例如由Google Play运行。回想一下，MAMADROID由不同的阶段组成，因此在下面，我们将评估每个阶段所产生的计算开销，以评估在实际部署中的可行性。我们在配备40个核心的2.30GHz CPU和128GB RAM的桌面电脑上运行实验，但只使用一个核心和分配16GB RAM进行评估。MAMADROID的第一步涉及从apk中提取调用图，这个任务的复杂性在不同应用之间差异很大。对于我们的恶意软件样本，平均完成时间为9.2秒±14（最短0.02秒，最长13分钟）。良性应用通常生成较大的调用图，并且提取它们的平均时间为25.4秒±63（最短0.06秒，最长18分钟）每个应用。请注意，我们的评估中不包括我们无法成功提取调用图的应用。接下来，我们测量提取调用序列并进行家族或包的抽象所需的时间，具体取决于MAMADROID的操作模式。在家族模式下，这个阶段的平均完成时间约为1.3秒（最长11.0秒），对于恶意和良性样本都是如此。抽象到包的时间略长一些，这是因为MAMADROID中使用了341个包。恶意应用的平均提取时间为1.67秒±3.1，良性样本为1.73秒±3.2。可以看出，在包模式下，调用序列的提取时间并没有显著增加。MAMADROID的第三步包括Markov链建模和特征向量提取。这个阶段的速度不管操作模式和数据集如何，都是很快的。具体来说，对于恶意样本，家族和包的平均时间分别为0.2秒±0.3和2.5秒±3.2（最长2.4秒和22.1秒），而对于良性样本，平均时间分别上升到0.6秒±0.3和6.7秒±3.8（最长1.7秒和18.4秒）。

does not take significantly more than in family mode. MAMADROID's third step includes Markov chain modeling and feature vector extraction. This phase is fast regardless of the mode of operation and datasets used. Specifically, with malicious samples, it takes on average 0.2s±0.3 and 2.5s±3.2 (and at most 2.4s and 22.1s), respectively, with families and packages, whereas, with benign samples, averages rise to 0.6s±0.3 and 6.7s±3.8 (at most 1.7s and 18.4s).

F. Runtime Performance Part-2

Finally, the last step involves classification, and performance depends on both the machine learning algorithm employed and the mode of operation. More specifically, running times are affected by the number of features for the app to be classified, and not by the initial dimension of the call graph, or by whether the app is benign or malicious. Regardless, in family mode, Random Forests, 1-NN, and 3-NN all take less than 0.01s. With packages, it takes, respectively, 0.65s, 1.05s, and 0.007s per app with 1-NN, 3-NN, Random Forests. Overall, when operating in family mode, malware and benign samples take on average, 10.7s and 27.3s respectively to complete the entire process, from call graph extraction to classification. Whereas, in package mode, the average completion times for malware and benign samples are 13.37s and 33.83s respectively. In both modes of operation, time is mostly (> 80%) spent on call graph extraction.

F. Runtime Performance

最后一步是分类，性能受到所采用的机器学习算法和操作模式的影响。具体来说，运行时间受到要分类的应用程序的特征数量的影响，而不受到调用图的初始维度或应用程序是良性还是恶意的影响。在家族模式下，随机森林、1-NN和3-NN的运行时间都小于0.01秒。使用包时，1-NN、3-NN和随机森林分别需要0.65秒、1.05秒和0.007秒来处理每个应用程序。

总体而言，在家族模式下，恶意样本和良性样本平均需要10.7秒和27.3秒来完成从调用图提取到分类的整个过程。而在包模式下，恶意样本和良性样本的平均完成时间分别为13.37秒和33.83秒。在两种操作模式下，时间大部分（>80%）耗费在调用图的提取上。

F. Runtime Performance Part-3

We also evaluate the runtime performance of DROIDAPIMINER [2]. Its first step, i.e., extracting API calls, takes 0.7s±1.5 (min 0.01s, max 28.4s) per app in our malware datasets. Whereas, it takes on average 13.2s±22.2 (min 0.01s, max 222s) per benign app. In the second phase, i.e., frequency and data flow analysis, it takes, on average, 4.2s per app. Finally, classification using 3-NN is very fast: 0.002s on average. Therefore, in total, DROIDAPIMINER takes respectively, 17.4s and 4.9s for a complete execution on one app from our benign and malware datasets, which while faster than MAMADROID, achieves significantly lower accuracy. In conclusion, our experiments show that our prototype implementation of MAMADROID is scalable enough to be deployed. Assuming that, everyday, a number of apps in the order of 10,000 are submitted to Google Play, and using the average execution time of benign samples in family (27.3s) and package (33.83s) modes, we estimate that it would take less than an hour and a half to complete execution of all apps submitted daily in both modes, with just 64 cores. Note that we could not find accurate statistics reporting the number of apps submitted everyday, but only the total number of apps on Google Play. 12 On average, this number increases of a couple of thousands per day, and although we do not know how many apps are removed, we believe 10,000 apps submitted every day is likely an upper bound.

F. Runtime Performance

我们还评估了DROIDAPIMINER [2]的运行时性能。它的第一步，即提取API调用，对我们的恶意软件数据集中的每个应用程序平均需要0.7秒±1.5（最小0.01秒，最大28.4秒）。而对于良性应用程序，平均需要13.2秒±22.2（最小0.01秒，最大222秒）。在第二阶段，即频率和数据流分析，平均每个应用程序需要4.2秒。最后，使用3-NN进行分类非常快速：平均0.002秒。因此，总的来说，DROIDAPIMINER在我们的良性和恶意软件数据集中分别需要17.4秒和4.9秒来完成一个应用程序的完整执行，虽然比MAMADROID更快，但准确率显著较低。

总结起来，我们的实验证明，我们对MAMADROID的原型实现足够可扩展，可以进行部署。假设每天向Google Play提交大约10,000个应用程序，并使用在家族（27.3秒）和包（33.83秒）模式下良性样本的平均执行时间，我们估计以64个核心，每天在两种模式下完成所有提交应用程序的执行时间将不到一个半小时。请注意，我们无法找到准确的统计数据来报告每天提交的应用程序数量，只能获得在Google Play上的应用程序总数。平均而言，该数字每天增加几千个，尽管我们不知道有多少应用程序被删除，但我们认为每天提交的10,000个应用程序很可能是一个上限。

V. DISCUSSION

We now discuss the implications of our results with respect to the feasibility of modeling app behavior using static analysis and Markov chains, discuss possible evasion techniques, and highlight some limitations of our approach.

V. DISCUSSION

我们现在就我们的研究结果讨论其对使用静态分析和马尔科夫链对应用程序行为进行建模的可行性的影响，并讨论可能的规避技术，并指出我们方法的一些局限性。

A. Lessons Learned

Our work yields important insights around the use of API calls in malicious apps, showing that, by modeling the sequence of API calls made by an app as a Markov chain, we can successfully capture the behavioral model of that app. This allows MAMADROID to obtain high accuracy overall, as well as to retain it over the years, which is crucial due to the continuous evolution of the Android ecosystem. As discussed in Section III, the use of API calls changes over time, and in different ways across malicious and benign samples. From our newer datasets, which include samples up to Spring 2016 (API level 23), we observe that newer APIs introduce more packages, classes, and methods, while also deprecating some. Fig. 6, 7(a), and 7(b) show that benign apps are using more calls than malicious ones developed around the same time. We also notice an interesting trend in the use of Android and Google APIs: malicious apps follow the same trend as benign apps in the way they adopt certain APIs, but with a delay of some years. This might be a side effect of Android malware authors' tendency to repackage benign apps, adding their malicious functionalities onto them.

A. Lessons Learned

我们的研究在恶意应用程序中使用API调用方面提供了重要的见解，表明通过将应用程序的API调用序列建模为马尔可夫链，我们可以成功捕捉该应用程序的行为模型。这使得MAMADROID能够获得整体上较高的准确性，并且能够在多年间保持这种准确性，这对于不断演变的Android生态系统来说非常关键。

正如第三节讨论的那样，API调用的使用会随着时间的推移而发生变化，并且在恶意和良性样本之间有不同的方式。从我们的最新数据集中，包括截至2016年春季（API级别23）的样本，我们观察到新的API引入了更多的包、类和方法，同时也废弃了一些。图6、7(a)和7(b)显示，良性应用程序在相同时间开发的恶意应用程序中使用的调用比恶意应用程序要多。我们还注意到Android和谷歌API的使用存在一个有趣的趋势：恶意应用程序在采用某些API的方式上与良性应用程序遵循相同的趋势，但延迟几年。这可能是Android恶意软件作者将恶意功能添加到良性应用程序中的副作用。

鉴于Android框架的频繁变化和恶意软件的不断演变，像DROIDAPIMINER [2]这样依赖于特定API调用是否存在或使用的系统在时间上变得越来越不有效。正如表III所示，使用训练集中使用的API调用之后发布的API调用的恶意软件无法被这些系统识别。相反，正如图11和12所示，MAMADROID可以检测到比训

Given the frequent changes in the Android framework and the continuous evolution of malware, systems like DROIDAPIMINER [2] -being dependent on the presence or the use of certain API calls -become increasingly less effective with time. As shown in Table III, malware that uses API calls released after those used by samples in the training set cannot be identified by these systems. On the contrary, as shown in Fig. 11 and12, MAMADROID detects malware samples that are 1 year newer than the training set obtaining an 86% Fmeasure (as opposed to 46% with DROIDAPIMINER). After 2 years, the value is still at 75% (42% with DROIDAPIMINER), dropping to 51% after 4 years. We argue that the effectiveness of MAMADROID's classification remains relatively high "over the years" owing to Markov models capturing app behavior. These models tend to be more robust to malware evolution because abstracting to families or packages makes the system less susceptible to the introduction of new API calls. Abstraction allows MAMADROID to capture newer classes/methods added to the API, since these are abstracted to already-known families or packages. In case newer packages are added to the API, and these packages start being used by malware, MAMADROID only requires adding a new state to the Markov chains, and probabilities of a transition from a state to this new state in old apps would be 0. Adding only a few nodes does not likely alter the probabilities of the other 341 nodes, thus, two apps created with the same purpose will not strongly differ in API calls usage if they are developed using almost consecutive API levels. We also observe that abstracting to packages provides a slightly better tradeoff than families. In family mode, the system is lighter and faster, and actually performs

练集更新一年的恶意软件样本，获得86%的F-measure（与DROIDAPIMINER的46%相比）。经过2年，该值仍然为75%（与DROIDAPIMINER的42%相比），经过4年后下降至51%。

我们认为MAMADROID分类的有效性在多年间保持相对较高，这要归功于马尔可夫模型捕获应用程序行为。这些模型对于恶意软件的演变具有更强的鲁棒性，因为将调用抽象为族或包使系统对新API调用的引入较不敏感。抽象使MAMADROID能够捕捉到添加到API中的新类/方法，因为这些被抽象为已知的族或包。如果API中添加了新的包，并且这些包开始被恶意软件使用，MAMADROID只需要向马尔可夫链中添加一个新状态，并且旧应用程序中从一个状态到该新状态的转换的概率将为0。仅添加少数节点不太可能改变其他341个节点的概率，因此，如果两个具有相同目的的应用程序几乎同时使用几乎连续的API级别进行开发，它们在API调用使用上不会有太大差异。

我们还观察到，对包进行抽象提供了比对族进行抽象更好的折衷方案。在族模式下，系统更轻量且更快，并且实际上在训练和测试集样本之间有两年以上的时间间隔时表现更好。然而，尽管两种操作模式都能有效检测到恶意软件，但将抽象到包级别的结果更好。然而，这并不意味着越少的抽象就越好：实际上，一个过于精细的系统除了产生难以承受的复杂性外，还很可能创建具有低概率转换的马尔可夫模型，最终导致分类的准确性降低。我们还强调，应用主成分分析（PCA）是一种保持高准确性并同时降低复杂度的良好策略。

better when there are more than two years between training and test set samples However, even though both modes of operation effectively detect malware, abstracting to packages yields better results overall. Nonetheless, this does not imply that less abstraction is always better: in fact, a system that is too granular, besides incurring untenable complexity, would likely create Markov models with low-probability transitions, ultimately resulting in less accurate classification. We also highlight that applying PCA is a good strategy to preserve high accuracy and at the same time reducing complexity.

B. Evasion

Next, we discuss possible evasion techniques and how they can be addressed. One straightforward evasion approach could be to repackage a benign app with small snippets of malicious code added to a few classes. However, it is difficult to embed malicious code in such a way that, at the same time, the resulting Markov chain looks similar to a benign one. For instance, our running example from Section II (malware posing as a memory booster app and executing unwanted commands as root) is correctly classified by MAMADROID; although most functionalities in this malware are the same as the original app, injected API calls generate some transitions in the Markov chain that are not typical of benign samples. The opposite procedure -i.e., embedding portions of benign code into a malicious app -is also likely ineffective against MAMADROID, since, for each app, we derive the feature vector from the transition probability between calls over the entire app. In other words, a malware developer would have to

B. Evasion

接下来，我们将讨论可能的逃避技术以及如何应对它们。一种直接的逃避方法可能是将一个良性应用重新打包，并在几个类中添加少量恶意代码。然而，很难以一种既能使得生成的马尔可夫链看起来与良性链类似又能嵌入恶意代码的方式。例如，我们在第二节的示例中，MAMADROID可以正确地将伪装成内存增加器应用并在root权限下执行不必要命令的恶意软件进行分类。尽管此恶意软件的大部分功能与原始应用相同，注入的API调用会生成一些在良性样本中不典型的马尔可夫链转换。

相反的情况——即在恶意应用中嵌入良性代码片段——对于MAMADROID而言也很有可能失效，因为我们根据整个应用程序的调用之间的转换概率推导特征向量。换句话说，恶意软件开发者需要以某种方式将良性代码嵌入恶意软件中，使得整体调用序列产生类似于良性应用中的转换概率，但这很难实现，因为如果调用序列必须不同（否则就没有攻击），那么模型也将不同。

攻击者还可以尝试从头创建一个具有与良性应用相似马尔可夫链的应用。由于这马尔可夫链是从应用程序中抽象的API调用序列派生出来的，实际上很难创建出产生与良性应用相似的马尔可夫链的序列，同时仍可以进行恶意行

embed benign code inside the malware in such a way that the overall sequence of calls yields similar transition probabilities as those in a benign app, but this is difficult to achieve because if the sequences of calls have to be different (otherwise there would be no attack), then the models will also be different. An attacker could also try to create an app from scratch with a similar Markov chain to that of a benign app. Because this is derived from the sequence of abstracted API calls in the app, it is actually very difficult to create sequences resulting in Markov chains similar to benign apps while, at the same time, actually engaging in malicious behavior. Nonetheless, in future work, we plan to systematically analyze the feasibility of this strategy. Moreover, attackers could try using reflection, dynamic code loading, or native code [42]. Because MAMADROID uses static analysis, it fails to detect malicious code when it is loaded or determined at runtime. However, MAMADROID can detect reflection when a method from the reflection package (java.lang.reflect) is executed. Therefore, we obtain the correct sequence of calls up to the invocation of the reflection call, which may be sufficient to distinguish between malware and benign apps. Similarly, MAMADROID can detect the usage of class loaders and package contexts that can be used to load arbitrary code, but it is not able to model the code loaded; likewise, native code that is part of the app cannot be modeled, as it is not Java and is not processed by Soot. These limitations are not specific of MAMADROID, but are a problem of static analysis in general, which can be mitigated by using MAMADROID alongside dynamic analysis techniques. Malware developers might also attempt to evade MA-MADROID by naming their

为。然而，在今后的工作中，我们计划系统分析这种策略的可行性。

此外，攻击者可以尝试使用反射、动态代码加载或本地代码[42]。由于MAMADROID使用静态分析，它在加载或运行时确定恶意代码时无法检测到。然而，当执行了来自反射包（java.lang.reflect）的方法时，MAMADROID可以检测到反射。因此，我们可以获得从调用到反射调用的正确调用序列，这可能足以区分恶意软件和良性应用。类似地，MAMADROID可以检测到可以用来加载任意代码的类加载器和包上下文的使用，但无法建模加载的代码；同样，在应用中的本地代码无法建模，因为它不是Java，也不会被Soot处理。这些限制不是特定于MAMADROID，而是静态分析的一般问题，可以通过将MAMADROID与动态分析技术结合使用来缓解。

恶意软件开发者还可能尝试通过将自定义程序包命名得与Android、Java或Google APIs的程序包相似，例如创建名称类似于java.lang.reflect.malware和java.lang.malware的包，以混淆MAMADROID并将其抽象为java.lang.reflect和java.lang。然而，通过将Android、Java或Google APIs的程序包列入白名单，可以轻松防止这种情况发生。

另一种方法可能是使用动态分派，即在包A中创建一个类X，使其扩展包B中的类Y，并且静态分析报告调用Y中定义的root()作为X.root()，但在运行时执行的是Y.root()。然而，通过稍微增加MAMADROID的计算成本，我们可以解决这个问题，方法是跟踪自定义类，这些类扩展或实现了已被识别的API中的类，并将此自定义类的多态函数抽象为相应的识别包中，同时将其自定义的覆盖函数抽象为自定义的类。

最后，可以使用标识符交错和其他混淆形式来混淆代码并隐藏恶意行为。然而，由于Android框架中的类不能通过混淆工具进行混淆，恶意软件开发者只能对自定义类进行混淆。MAMADROID将混淆的调用标记为混淆，因此，最终这些内容将在应用的行为模型（和马尔可夫链）中被捕获。在我们的样本

self-defined packages in such a way that they look similar to that of the android, java, or google APIs, e.g., creating packages like java.lang.reflect.malware and java.lang.malware, aiming to confuse MAMADROID into abstracting them to respectively, java.lang.reflect and java.lang. However, this is easily prevented by whitelisting the list of packages from android, java, or google APIs. Another approach could be using dynamic dispatch so that a class X in package A is created to extend class Y in package B with static analysis reporting a call to root() defined in Y as X.root(), whereas, at runtime Y.root() is executed. This can be addressed, however, with a small increase in MAMADROID's computational cost, by keeping track of self-defined classes that extend or implement classes in the recognized APIs, and abstract polymorphic functions of this self-defined class to the corresponding recognized package, while, at the same time, abstracting as self-defined overridden functions in the class. Finally, identifier mangling and other forms of obfuscation could be used aiming to obfuscate code and hide malicious actions. However, since classes in the Android framework cannot be obfuscated by obfuscation tools, malware developers can only do so for self-defined classes. MAMADROID labels obfuscated calls as obfuscated so, ultimately, these would be captured in the behavioral model (and the Markov chain) for the app. In our sample, we observe that benign apps use significantly less obfuscation than malicious apps, indicating that obfuscating a significant number of classes is not a good evasion strategy since this would likely make the sample more easily identifiable as malicious.

中，我们观察到良性应用使用的混淆比恶意应用少得多，这表明混淆大量类不是一个好的逃避策略，因为这很可能会使样本更容易被识别为恶意。

C. Limitations

MAMADROID requires a sizable amount of memory in order to perform classification, when operating in package mode, working on more than 100,000 features per sample. The quantity of features, however, can be further reduced using feature selection algorithms such as PCA. As explained in Section IV when we use 10 components from the PCA the system performs almost as well as the one using all the features; however, using PCA comes with a much lower memory complexity in order to run the machine learning algorithms, because the number of dimensions of the features space where the classifier operates is remarkably reduced. Soot [52], which we use to extract call graphs, fails to analyze some apks. In fact, we were not able to extract call graphs for a fraction (4.6%) of the apps in the original datasets due to scripts either failing to apply the jb phase, which is used to transform Java bytecode to the primary intermediate representation (i.e., jimple) of Soot or not able to open the apk. Even though this does not really affect the results of our evaluation, one could avoid it by using a different/custom intermediate representation for the analysis or use different tools to extract the call graphs. In general, static analysis methodologies for malware detection on Android could fail to capture the runtime environment context, code that is executed more frequently, or other effects stemming from user input [5]. These limitations can be addressed using dynamic analysis, or by recording function calls on a device. Dynamic analysis observes the live performance of the samples, recording what activity is actually performed at runtime. Through dynamic analysis, it is

C. Limitations

C. Limitations:

MAMADROID在执行分类时，需要大量的内存，特别是在包级别操作时，每个样本要处理超过100,000个特征。然而，使用主成分分析（PCA）等特征选择算法可以进一步减少特征数量。正如第IV节所解释的那样，当我们使用来自PCA的10个主成分时，系统的性能几乎与使用所有特征时相当；然而，使用PCA可以大大降低内存复杂性，以便运行机器学习算法，因为分类器所操作的特征空间的维数大幅减少。

我们使用的Soot [52]用于提取调用图，但有时会无法分析某些apk文件。事实上，我们无法为原始数据集中约4.6%的应用程序提取调用图，原因是脚本无法应用jb阶段，用于将Java字节码转换为Soot 的主要中间表示（即jimple），或无法打开apk文件。尽管这对于我们评估结果并没有实质影响，但可以通过使用不同的/自定义中间表示进行分析或使用不同的工具来提取调用图，可以避免这种情况。

总的来说，在Android上用于检测恶意软件的静态分析方法可能无法捕捉运行环境上下文、更频繁执行的代码或源于用户输入的其他效果[5]。这些限制可以通过使用动态分析或在设备上记录函数调用来解决。动态分析观察样本的实时性能，记录运行时实际执行的活动。通过动态分析，还可以为应用程序提供输入，并分析应用程序对这些输入的反应，超越静态分析的限制。为此，我们计划在未来的工作中整合动态分析，以构建MAMADROID使用的模型。

also possible to provide inputs to the app and then analyze the reaction of the app to these inputs, going beyond static analysis limits. To this end, we plan to integrate dynamic analysis to build the models used by MAMADROID as part of future work.

VI. RELATED WORK

Over the past few years, Android security has attracted a wealth of work by the research community. In this section, we review (i) program analysis techniques focusing on general security properties of Android apps, and then (ii) systems that specifically target malware on Android.

VI. RELATED WORK

在过去几年里，Android安全性已经引起了研究界的广泛关注。本节中，我们将回顾(i)关注Android应用程序一般安全属性的程序分析技术，然后再回顾(ii)专门针对Android上的恶意软件的系统。

A. Program Analysis

Previous work on program analysis applied to Android security has used both static and dynamic analysis. With the former, the program's code is decompiled in order to extract features without actually running the program, usually employing tools such as Dare [41] to obtain Java bytecode. The latter involves real-time execution of the program, typically in an emulated or protected environment. Static analysis techniques include work by Felt et al. [21], who analyze API calls to identify over-privileged apps, while Kirin [20] is a system that examines permissions requested by apps to perform a lightweight certification, using a set of security rules that indicate whether or not the security configuration bundled with the app is safe. RiskRanker [28] aims to identify zero-day Android malware by

A. Program Analysis

在Android安全领域，之前的程序分析工作采用了静态分析和动态分析的方法。在静态分析中，程序的代码被反编译以提取特征，而不需要实际运行程序，通常使用Dare [41]等工具获取Java字节码。而动态分析则涉及对程序的实时执行，通常在模拟或受保护的环境中进行。

静态分析技术包括Felt等人的工作[21]，他们通过分析API调用来识别过度授权的应用；而Kirin [20]是一个系统，通过检查应用程序请求的权限进行轻量级认证，使用一组安全规则来指示应用程序捆绑的安全配置是否安全。RiskRanker [28]旨在通过评估由不受信任的应用程序引起的潜在安全风险来识别零日Android恶意软件。它筛选了来自Android市场的大量应用程序，并检查它们以检测某些行为，例如加密和动态代码加载，这些行为形成恶意模式，并可用于检测隐蔽的恶意软件。其他方法，如CHEX [37]，使用数据流分析自动审核Android应用程序的漏洞。静态分析还被

assessing potential security risks caused by untrusted apps. It sifts through a large number of apps from Android markets and examines them to detect certain behaviors, such as encryption and dynamic code loading, which form malicious patterns and can be used to detect stealthy malware. Other methods, such as CHEX [37], use data flow analysis to automatically vet Android apps for vulnerabilities. Static analysis has also been applied to the detection of data leaks and malicious data flows from Android apps [6], [34], [35], [62]. DroidScope [59] and TaintDroid [19] monitor run-time app behavior in a protected environment to perform dynamic taint analysis. DroidScope performs dynamic taint analysis at the machine code level, while TaintDroid monitors how third-party apps access or manipulate users' personal data, aiming to detect sensitive data leaving the system. However, as it is unrealistic to deploy dynamic analysis techniques directly on users' devices, due to the overhead they introduce, these are typically used offline [45], [50], [67]. ParanoidAndroid [44] employs a virtual clone of the smartphone, running in parallel in the cloud and replaying activities of the device -however, even if minimal execution traces are actually sent to the cloud, this still takes a non-negligible toll on battery life. Recently, hybrid systems like IntelliDroid [56] have also been proposed that use input generators, producing inputs specific to dynamic analysis tools. Other work combining static and dynamic analysis include [8], [25], [31], [58].

应用于检测来自Android应用程序的数据泄露和恶意数据流[6]，[34]，[35]，[62]。

DroidScope [59]和TaintDroid [19]在受保护环境中监视运行时应用程序行为以进行动态污点分析。DroidScope 在机器代码级别执行动态污点 分析，而TaintDroid 监控第三方应用程序如何访问或操作用户个人数据，以检测敏感数据是否离开系统。然而，由于动态分析方法引入的额外开销，直接在用户设备上部署动态分析技术是不现实的，因此这些方法通常是离线使用的[45]，[50]，[67]。ParanoidAndroid [44]使用在云端平行运行的智能手机虚拟克隆，重放设备的活动 - 然而，尽管实际上只有最小的执行跟踪被发送到云端，但这仍然对电池寿命产生了非可忽略的影响。

最近，还提出了混合系统，如IntelliDroid [56]，它使用输入生成器生成特定于动态分析工具的输入。其他结合静态和动态分析的工作包括[8]，[25]，[31]，[58]。

B. Android Malware Detection Part-1

B. Android Malware Detection

A number of techniques have used signatures for Android malware detection. NetworkProfiler [18] generates network profiles for Android apps and extracts fingerprints based on such traces, while work in [12] obtains resource-based metrics (CPU, memory, storage, network) to distinguish malware activity from benign one. Chen et al. [15] extract statistical features, such as permissions and API calls, and extend their vectors to add dynamic behavior-based features. While their experiments show that their solution outperforms, in terms of accuracy, other antivirus systems, Chen et al. [15] indicate that the quality of their detection model critically depends on the availability of representative benign and malicious apps for training. Similarly, ScanMe Mobile [64] uses the Google Cloud Messaging Service (GCM) to perform static and dynamic analysis on apks found on the device's SD card. The sequences of system calls have also been used to detect malware in both desktop and Android environments. Hofmeyr et al. [30] demonstrate that short sequences of system calls can be used as a signature to discriminate between normal and abnormal behavior of common UNIX programs. Signaturebased methods, however, can be evaded using polymorphism and obfuscation, as well as by call re-ordering attacks [36], even though quantitative measures, such as similarity analysis, can be used to address some of these attacks [48]. MAMADROID inherits the spirit of these approaches, proposing a statistical method to model app behavior that is more robust against evasion attempts. In the Android context, Canfora et al. [11] use the sequences of three system calls (extracted from the execution traces of apps under analysis) to detect malware. This approach models specific malware families, aiming to

许多技术在Android恶意软件检测中使用了签名。 NetworkProfiler [18]为Android应用程序生成网络配置文件，并基于这些迹象提取指纹，而[12]中的研究通过获取基于资源的度量指标（CPU、内存、存储、网络）来区分恶意活动和良性活动。陈等人[15]提取权限和API调用等统计特征，并扩展它们的向量以添加动态基于行为的特征。尽管他们的实验表明，他们的解决方案在准确性上超过了其他杀毒软件系统，但陈等人[15]指出，他们的检测模型的质量严重依赖于具有代表性的良性和恶意应用程序用于训练。类似地，ScanMe Mobile [64]使用Google Cloud Messaging Service（GCM）对设备SD卡上的apk进行静态和动态分析。系统调用序列还用于检测桌面和Android环境中的恶意软件。 Hofmeyr等人[30]证明了可以使用系统调用的短序列作为签名来区分常见UNIX程序的正常行为和异常行为。然而，基于签名的方法可以通过多态性和混淆来逃避检测，以及通过调用重新排序攻击[36]，尽管定量措施，如相似性分析，可以用于解决某些这些攻击[48]。MAMADROID继承了这些方法的精神，提出了一种对抗逃避尝试更加强大的应用行为建模统计方法。 在Android环境中，Canfora等人[11]使用三个系统调用的序列（从分析应用程序的执行跟踪中提取）来检测恶意软件。这种方法模型化了特定的恶意软件家族，旨在识别属于这些家族的其他样本。相比之下，MAMADROID的目标是检测之前未见过的恶意软件，并且我们还展示了我们的系统可以检测到即使在系统训练后多年出现的新的恶意软件家族。此外，使用严格的系统或API调用序列可以很容易地被恶意软件作者躲避，他们可以添加不必要的调用来有效地逃避检测。相反，MAMADROID构建了一个Android应用程序的行为模型，使其对这种类型的逃避具有鲁棒性。动态分析也被应用于通过使用在设备运行时将执行的通常输入的预定义脚本来检测Android恶意软件。然而，由于触发恶意行为的概率较低，这可能是不充分的，并且可以被有知识的对手规避，正如Wong和Lie [56]所建议的那样。其他方法包括随机模糊测试[38]，[63]和混合测试[3]，[26]。只有当恶意行为的代码在分析期间实际运行时，动态分析才能检测到恶意活动。此外，根据[54]，移动恶意软件作者经常使用模

identify additional samples belonging to such families. In contrast, MAMADROID's goal is to detect previously-unseen malware, and we also show that our system can detect new malware families that even appear years after the system has been trained. In addition, using strict sequences of system or API calls can be easily evaded by malware authors who could add unnecessary calls to effectively evade detection. Conversely, MAMADROID builds a behavioral model of an Android app, which makes it robust to this type of evasion. Dynamic analysis has also been applied to detect Android malware by using predefined scripts of common inputs that will be performed when the device is running. However, this might be inadequate due to the low probability of triggering malicious behavior, and can be side-stepped by knowledgeable adversaries, as suggested by Wong and Lie [56]. Other approaches include random fuzzing [38], [63] and concolic testing [3], [26]. Dynamic analysis can only detect malicious activities if the code exhibiting malicious behavior is actually running during the analysis. Moreover, according to [54], mobile malware authors often employ emulation or virtualization detection strategies to change malware behavior and eventually evade detection.

拟或虚拟化检测策略来更改恶意软件的行为并最终逃避检测。

B. Android Malware Detection Part-2

Aiming to complement static and dynamic analysis tools, machine learning techniques have also been applied to assist Android malware detection. Droidmat [57] uses API call tracing and manifest files to learn features for

B. Android Malware Detection

为了补充静态和动态分析工具，机器学习技术也被应用于辅助Android恶意软件检测。Droidmat [57] 使用API调用跟踪和清单文件来学习恶意软件检测的特征，而Gascon等人 [24] 则依赖于内嵌的调用图。Droid-Miner [60] 研究敏感的Android/Java框架API函数和

malware detection, while Gascon et al. [24] rely on embedded call graphs. Droid-Miner [60] studies the program logic of sensitive Android/Java framework API functions and resources, and detects malicious behavior patterns. MAST [14] statically analyzes apps using features such as permissions, presence of native code, and intent filters and measures the correlation between multiple qualitative data. Crowdroid [10] relies on crowdsourcing to distinguish between malicious and benign apps by monitoring system calls. AppContext [61] models security-sensitive behavior, such as activation events or environmental attributes, and uses SVM to classify these behaviors, while RevealDroid [23] employs supervised learning and obfuscation-resilient methods targeting API usage and intent actions to identify their families. DREBIN [5] automatically deduces detection patterns and identifies malicious software directly on the device, performing a broad static analysis. This is achieved by gathering numerous features from the manifest file as well as the app's source code (API calls, network addresses, permissions). Malevolent behavior is reflected in patterns and combinations of extracted features from the static analysis: for instance, the existence of both SEND_SMS permission and the android.hardware.telephony component in an app might indicate an attempt to send premium SMS messages, and this combination can eventually constitute a detection pattern. In Section IV, we have already introduced, and compared against, DROIDAPIMINER [2]. This system relies on the top-169 API calls that are used more frequently in the malware than in the benign set, along with data flow analysis on calls that are frequent in both benign and malicious apps, but occur up to 6% more in the latter. As shown in our evaluation, using

资源的程序逻辑，并检测恶意行为模式。MAST [14] 静态分析应用程序，使用权限、原生代码的存在以及意图过滤器等特征，并测量多个定性数据之间的相关性。Crowdroid [10] 通过监视系统调用，依赖于众包来区分恶意和良性应用。AppContext [61] 对安全敏感行为（例如激活事件或环境属性）进行建模，并使用支持向量机对这些行为进行分类，而RevealDroid [23] 则采用监督学习和对抗性抗混淆方法，针对API使用和意图动作识别其家族。DREBIN [5] 在设备上自动推断检测模式，并直接识别恶意软件，进行广泛的静态分析。这是通过从清单文件和应用程序的源代码（API调用、网络地址、权限）中收集大量特征来实现的。从静态分析中提取的特征的模式和组合反映了恶意行为：例如，一个应用程序中同时存在SEND_SMS权限和android.hardware.telephony组件可能表明试图发送高级短信，并且这种组合最终可以构成一个检测模式。

在第四节中，我们已经介绍并与DROIDAPIMINER [2]进行了比较。该系统依赖于在恶意软件中使用频率高于良性集合的前169个API调用，并对良性和恶意应用程序中频繁出现但后者增加了6%的调用进行数据流分析。正如我们的评估所示，在训练期间使用最常见的调用需要不断进行重新训练，因为恶意软件和Android API都在不断演化。相反，MAMADROID可以有效地对良性和恶意的Android应用进行建模，并对它们进行高效的分类。与DROIDAPIMINER相比，我们的方法对Android框架的变化更具弹性，从而减少了重新训练分类器的频率。

the most common calls observed during training requires constant retraining, due to the evolution of both malware and the Android API. On the contrary, MAMADROID can effectively model both benign and malicious Android apps, and perform an efficient classification on them. Compared to DROIDAPIMINER, our approach is more resilient to changes in the Android framework than DROIDAPIMINER, resulting in a less frequent need to re-train the classifier.

B. Android Malware Detection Part-3

Overall, compared to state-of-the-art systems like DREBIN [5] and DROIDAPIMINER [2], MAMADROID is more generic and robust as its statistical modeling does not depend on specific app characteristics, but can actually be run on any app created for any Android API level. Finally, also related to MAMADROID are Markov-chain based models for Android malware detection. Chen et al. [16] dynamically analyze system-and developer-defined actions from intent messages (used by app components to communicate with each other at runtime), and probabilistically estimate whether an app is performing benign or malicious actions at run time, but obtain low accuracy overall. Canfora et al. [13] use a Hidden Markov model (HMM) to identify malware samples belonging to previously observed malware families, whereas, MAMADROID can detect previously unseen malware, not relying on specific malware families.

B. Android Malware Detection

总的来说，与DREBIN [5]和DROIDAPIMINER [2]等先进系统相比，MAMADROID更加通用和健壮，因为它的统计建模不依赖于特定的应用程序特征，实际上可以在为任何Android API级别创建的任何应用程序上运行。最后，与MAMADROID相关的还有基于马尔可夫链的模型用于Android恶意软件检测。Chen等人[16]从意图消息中动态分析系统和开发者定义的操作（用于应用程序组件在运行时进行通信），并在运行时概率地估计一个应用程序是否执行良性或恶意操作，但总体准确率较低。Canfora等人[13]使用隐马尔可夫模型（HMM）来识别属于先前观察到的恶意软件家族的恶意软件样本，而MAMADROID可以检测以前未见过的恶意软件，不依赖于特定的恶意软件家族。

VII. CONCLUSION

VII. CONCLUSION

This paper presented MAMADROID, an Android malware detection system based on modeling the sequences of API calls as Markov chains. Our system is designed to operate in one of two modes, with different granularities, by abstracting API calls to either families or packages. We ran an extensive experimental evaluation using, to the best of our knowledge, the largest malware dataset ever analyzed in an Android malware detection research paper, and aiming at assessing both the accuracy of the classification (using F-measure, precision, and recall) and runtime performances. We showed that MAMADROID effectively detects unknown malware samples developed earlier or around the same time as the samples on which it is trained (F-measure up to 99%). It also maintains good detection performance: one year after the model has been trained the F-measure value is 87%, and after two years it is 73%. We compared MAMADROID to DROIDAPIMINER [2], a state-of-the-art system based on API calls frequently used by malware, showing that, not only does MAMADROID outperforms DROIDAPIMINER when trained and tested on the same datasets, but that it is also much more resilient over the years to changes in the Android API. Overall, our results demonstrate that the type of statistical behavioral models introduced by MAMADROID are more robust than traditional techniques, highlighting how our work can form the basis of more advanced detection systems in the future. As part of future work, we plan to further investigate the resilience to possible evasion techniques, focusing on repackaged malicious apps as well as injection of API calls to maliciously alter Markov models. We also plan to explore the use of finer-grained abstractions as well as the possibility to seed the behavioral modeling performed by

本文介绍了一个基于将API调用序列建模为马尔可夫链的Android恶意软件检测系统——MAMADROID。我们的系统设计有两种模式，以不同的粒度将API调用抽象为家族或包。我们进行了广泛的实验评估，使用了迄今为止在Android恶意软件检测研究论文中最大的恶意软件数据集，并旨在评估分类的准确性（使用F-measure、精确度和召回率）和运行时性能。我们展示了MAMADROID能够有效地检测先前开发或与训练样本时间相近的未知恶意软件样本（F-measure高达99%）。它还保持良好的检测性能：模型训练一年后，F-measure值为87%，两年后为73%。

我们将MAMADROID与基于恶意软件常用API调用的当前最先进系统DROIDAPIMINER进行了比较，结果表明，不仅在相同数据集上训练和测试时MAMADROID优于DROIDAPIMINER，而且在多年间对Android API变化更具弹性。总体而言，我们的结果表明，MAMADROID引入的统计行为模型比传统技术更加稳健，突显了我们的工作如何成为未来更先进的检测系统的基础。作为未来的工作的一部分，我们计划进一步研究对可能的规避技术的弹性，重点关注重新封装的恶意应用程序以及注入API调用以恶意更改马尔可夫模型的情况。我们还计划探索使用更细粒度的抽象以及使用动态分析而不是静态分析来启动MAMADROID执行的行为建模。由于数据的较大规模，我们没有将其公开在线获取，但可以根据要求获取数据集和特征向量。

MAMADROID with dynamic instead of static analysis. Due to the large size of the data, we have not made them readily available online but both the datasets and the feature vectors can be obtained upon request.

Acknowledgments. We wish to thank the anonymous reviewers for their feedback, our shepherd Amir Houmansadr for his help in improving our paper, and Yousra Aafer for kindly sharing the DROIDAPIMINER source code with us. We also wish to thank Yanick Fratantonio for his comments on an early draft of the paper. This research was supported by the EPSRC under grant EP/N008448/1, by an EPSRC-funded "Future Leaders in Engineering and Physical Sciences" award, a Xerox University Affairs Committee grant, and by a small grant from GCHQ. Enrico Mariconti was supported by the EPSRC under grant 1490017, while Lucky Onwuzurike was funded by the Petroleum Technology Development Fund (PTDF).

致谢。我们要感谢匿名评审人员对我们的反馈意见，我们的指导老师Amir Houmansadr对我们的论文改进提供的帮助，以及Yousra Aafer友好地与我们共享DROIDAPIMINER的源代码。我们还要感谢Yanick Fratantonio对论文早期草稿的意见。这项研究得到了EPSRC的EP/N008448/1号授予的资助，以及EPSRC资助的“未来工程和物理科学领域的领军人才”奖、施乐大学事务委员会资助、以及GCHQ提供的一个小额资助。Enrico Mariconti得到了EPSRC的1490017号授予的支持，而Lucky Onwuzurike得到了石油技术发展基金(PTDF)的资助。